Loading JSON, data processing, interactivity

Programming Foundations – Fall 2022

rev. 2022.11.29.b

JSON: JavaScript Object Notation

Common Data Exchange Formats

- Images and other complex media are usually sent/received in binary formats
  - not very readable by humans because binary
  - Examples: **jpeg**, **mp3**, **mp4**
- Numeric and text data is often sent/received in a text format
  - often readable by humans
  - Examples: **csv**, **json**
- Any of these formats can be used to exchange data between client (web browser) and server or to load and save data on your local computer.
  - You can also make up your own file formats and write your own parsers (interpreters to get your data in and out of your format), but starting from an existing format is usually easier… and JSON is a format that works for almost any need!

Use JSON to load and save JavaScript **Data** Objects (Properties, not Methods)

- JavaScript Object Notation can store any type of JavaScript data (information, not code that takes action on the information) that we've explored so far
- These JavaScript Objects can be as complex or simple as needed. Some examples as you've seen them in JavaScript:
  - Simple Array: `[1,2,3]`
  - an Object storing multiple Arrays that in turn store multiple Objects:

```
{list1: [{subitem1:"A"},{subitem2:"B", subitem3: "C"}],
 list2: [{subitem1:"C"},{subitem2:"D"}],
 list3: [{subitem1:"A"},{subitem2:"E"}]}
```

- JSON saves the data in almost the same format that you're used to seeing in JavaScript.

Another format, JSONP, allows saving JavaScript code (methods) as well as data. However, use of this format requires creating security exceptions, so it is used far less than JSON.

"key" = property name
"value" = value stored in property

- The combination of a JavaScript Object property and its value is called a **key/value pair**
- When writing about your code, you can usually use the terms "key" and "property" interchangeably and people will know what you mean!
  - Similar to **variables** vs. **properties** and **functions** vs. **methods**. We want you to use terminology that shows you understand what you're doing, but we won't nitpick on these finer points!
- When **reading** JavaScript language explanations, you will most often encounter the term "key/value pair", so remember what it means!
- This terminology is even built in to JavaScript and D3!
  - You may recall that we used **.keys()** to extract unique values from our example city data
  - This is another instance in which you will need to use the correct term; JavaScript is much pickier than your instructors!

JSON requires **quotation marks** around every **key** (property name) at every level

JavaScript Object

```
let myData = {
list1:[
    {subitem1:"A"},
    {subitem2a: [3,4],
     subitem2b: "C"}
    ],
list2:[
    {subitem1: {a:7, b:true}},
    {subitem2:"D"}
    ]
}
```

JSON Encoding

```
{
"list1":[
    {"subitem1":"A"},
    {"subitem2a": [3,4],
     "subitem2b": "C"}
    ],
"list2":[
    {"subitem1":{"a":7, "b":true}},
    {"subitem2":"D"}
    ]
}
```

Otherwise, it's basically identical to creating a JavaScript Object!

JSON file format does not support **code commenting** or **variable assignment**

Valid JavaScript Object

```
let myData = {
list1:[
    {subitem1:"A"},
    {subitem2a: [3,4],
     subitem2b: "C"}
    ], // end list1
list2:[
    {subitem1: {a:7, b:true}},
    {subitem2:"D"}
    ] // end list2
} // end myData
```

Valid JSON Encoding

```
{
"list1":[
    {"subitem1":"A"},
    {"subitem2a": [3,4],
     "subitem2b": "C"}
    ],
"list2":[
    {"subitem1":{"a":7, "b":true}},
    {"subitem2":"D"}
    ]
}
```

Just remember to remove these extra things when converting your Objects to JSON.

Demo: JSON can only store values, not variable references

```
let luckyNumbers = [1, 2, 3];
let myObject = {name: "Fred", numbers: luckyNumbers};
luckyNumbers[0] = 7;
let myJSON = JSON.stringify(myObject);
luckyNumbers[2] = 9;
let myRestoredObject = JSON.parse(myJSON);
luckyNumbers[1] = 8;
console.log(myObject.numbers);
console.log(myRestoredObject.numbers);
```

myObject.numbers contains a reference

change specific Array element

store a copy of myObject as JSON

change specific Array element

parse JSON into a new object

change specific Array element

myObject.numbers is [7, 8, 9]

myRestoredObject.numbers is [7, 2, 3]

JSON.stringify() "freezes" the values so that no variable references remain in the stored data and every stored key has a fixed value! Any connection to the original data source is permanently broken.

JSON for Term Project

Starting this week, you'll put some or all of your data into JSON format in order to use it for your visualizations.

- Until the final version of your Term Project, you have the option to include only the data you need for your visualization in your JSON file(s).
  - In other words, you can manually restructure your data to fit the assignment, if that aids your understanding of what you're doing.
  - However, we recommend practicing techniques for manipulating your data as much as you can so that you will be prepared for the final term project requirement

For the final version of your Term Project, **all of your raw data* needs to end up in your JSON file**, even if much of it is transformed (or ignored) after being loaded by your JavaScript!

**Exception: You can exclude data that was inconsistently collected – i.e., data that you stopped collecting or changed collection methods partway through the project.*

Final Project Framework

- Download and open **term_project_framework.zip**
- Use this framework for Lab #10 and all future attempts at your term project
- Ensures you have all parts necessary for term project
- You can remove parts that you don't need yet
  - For final, you must remove any unused parts
- For all submissions using this framework, you must:
  - remove instructional comments
  - add your own explanatory comments for each major code section
- For the final term project submission, you must also write full function documentation for each major function in your program
  - This includes any of the ones that we provide in the framework that you use in your final!

index.html

- Be sure to read through all of the comments
- Delete comments after you have followed the instructions
  - You can always open another copy of the starter if you forget something!
- Provides the basic framework for your page, but you can customize layout
  - Be sure that your visualization is at top of page and any titles and text are below!
- Remember, **d3.min.js** and any **css** (which is optional) must load before the page **<body>**
  - Loading CSS early ensures that the page renders correctly
  - Loading D3 early ensures that it is ready when we reach the bottom of the page
- We have **main.js** load after the page body to ensure that everything else is loaded

main.css

- Includes some styles that we have used for other projects
- Using these is optional. You can replace this CSS with your own
  - Be sure to at least look through this CSS. Remember, if you use the class names provided in this file, anything assigned those classes will also take on these styles!
- Note the example /* code comment */
  - You should provide code comments in your CSS to explain how it is being used in your project.
- If you decide not to use an external CSS file, be sure to remove this file
  - Be sure to also remove the **<link>** tag from **index.html**

data.json

- A file containing the example JSON data from the heatmap exercise
  - We will also use this data in the Cleveland Dot Plot example
- If you are having trouble loading data from a file, you can temporarily just assign data to a variable, as you have been doing so far.
  - For your term project final, you will need to figure out how to make this work
  - For **attempt1/final**, you are not required to use this part of the framework
- Replace the data in this file with your own JSON-formatted data.
- You may also give this file a name that better describes your project, such as **myStudyHabits.json**
  - The file name should follow class file name conventions (no spaces!) and must have the extension **.json** so that it's read properly

d3.min.js

- As always, you do not need to look at this file... much
- You should, however, make sure that you are always using **v7.6.1**
  - You can find the version number in the code comment in line 1 of the file
  - This is the version that is required for the term project

main.js

- Sets up some of the usual **configuration variables** for you
  - Here we have created an Object called **margin** with four **properties** for the four borders.
    - This is the way that the creator of D3 usually works, but you can continue to use four separate variables if you prefer
- Provides an **svg** to draw to
  - Optionally provides an **svg group** stored in variable **viz** for you to draw to
    - You need to uncomment the code for this
    - It is up to you to decide how you prefer to set up your margins and offset your drawing
- The **async function()** code block is the one section of code that we won't ask you to explain. All you need to know about it is:
  - It loads **data.json** (or whatever name you give to the filename in quotes)
  - It **parses** the data into a JavaScript **object**
  - It passes that data object as a **parameter** to function **buildVisualization** (or whatever function name you provide)
  - It takes any data **return**ed from the function that it called and assigns it to global **data**

function buildVisualization(data)

- Function that receives the loaded JSON data
- Calls a series of "stub functions" that we've provided to keep you organized
  - You can change the function names or parameters on any of these
- Finally returns the **data** same as it was sent in, ensuring that global variable **data** will have a copy of the data in case you need it again later

"stub functions"

- Functions that have minimal information in them.
- More like descriptions of what the program should do at each phase
- **organizeData** should take your data and filter and sort it, if needed.
  - If you don't need this function, remove the call to it and remove the function
  - Note that this function is called **first** in our **buildVisualization** function, though it is written **second**in the framework. You can change the order of these functions if that makes more sense to you!
- **buildScales** should take the data sent to it and build things like **xScale** and **yScale**
  - Note that we have provided **global variables** for some scales that you would commonly use. Remember, any variable created within a function is **local**, so would have to be **passed** as a **parameter** to another function or **returned** from a function if it needed to be used elsewhere.
- **drawVisualization** should actually do the work of drawing your visualization, using everything that you've set up earlier
  - We have provided a suggested second parameter **drawing** that you can use to tell your function whether you want to draw to your **svg** or to your **viz** group or to some other DOM element that you set up!

Debugger within a code block

- Until now, you have been able to inspect your variables and check what's going on with your visualization after it has been drawn
- This is because everything that you have been doing has been in the **global context**
  - In other words, all of your important variables have been accessible via the JavaScript Console because they have not been created within functions
- When using the term project framework, most of the work is happening within functions
- We have provided you some globals, but if you want to investigate what's happening in your drawing code, you will need to use the debugger!
- The debugger can be intimidating, but it is really worth getting to know
- If nothing else, you can use the **debugger;** keyword to stop your program's execution in the middle of the code block where you're having a problem.
- Then, all of the **local variables** that exist in the current code block will still be available to you at the JavaScript Console, which might be a gentler way of inspecting them!

Cleveland Dot Plot Example: Simple Interactivity

cdp_framework_starter.zip

- Download, unZIP, and test
- Built on your term project framework, but with enhancements
  - Consider similar enhancements that will help you to understand your own project better. Don't just copy ours!

function **buildVisualization**(data)

- Receives the data loaded by **d3.json**
- Calls **organizeData**, passing along the data and getting it back in **renderData**
  - For your project, if you need to do any data transformation such as filtering or reducing or sorting, you might do this in **organizeData**
- Calls **buildScales** using the organized version of data
  - Could just use **data** as parameter if you didn't need to organize data at all
- Calls **drawVisualization**, passing the organized data and where to draw to
  - **viz** is actually a global variable in this program, but we're explicitly passing it here to make clear that it is possible to vary what **drawVisualization** draws to
- Finally, **return**s the data so that it can be stored in global **data**
  - Here, we know we will want to work with the data as we've organized it so we return **renderData**. We could just return **data** if we wanted to be able to work with our raw data
    - In this particular example, there is actually no difference, since we haven't manipulated our data yet!

function **drawVisualization**(data, drawing)

- Notice that we have chosen to break up our drawing into a couple of functions
- This function simply draws the X axis and Y axis and then calls another function **plotPoints()** to do the bulk of the drawing
- Why do this? We want to be able to redraw the temperature data for the month that the user has selected, but we don't need to redraw our entire visualization each time!
  - This is an advanced way of working that can show us you really understand what you're doing in your visualization (as long as you're not just copying this example, of course)
  - If you have trouble with this concept, it is also OK to redraw the whole visualization each time if you add any interactivity.
    - Remember D3's *selection*.**remove()** if you need to do this!
    - Also remember that interactivity is entirely optional on the term project. You should only include it if you feel that it will make your visualization clearer (or if your instructors have advised you to do so!)

Review Challenge #1: **organizeData()**

- Right now, there isn't really any order to our city names
- Function **organizeData()** currently just returns the data same as it is sent in
- Try to write a **sort()** function so that the data is alphabetized:
  - Chicago appearing at the top of our Y axis
  - San Jose appearing at the bottom of our Y axis

- Reminder of *Array*.**sort()** helper function form:

```
function (item1, item2) {
    if (condition for swapping) {
        return 1
    }
    return -1
}
```

Review Challenge #1: Solution

```
data.sort(function (a, b) {
    if (a.city.toLowerCase() > b.city.toLowerCase()) {
        return 1
    }
        return -1
})
```

- Note: Although we return **data** in order to follow the format of our framework, technically our global **data** has already been updated.
  - Why? Because *Array*.**sort()** sorts the data "in place" instead of making a copy of the Array!

Interactive Code

- At the bottom of **main.js** you will find the code that should make this project interactive
- The first part of this code connects the **"change"** event on our **<select>** menu (which we identify by **id="month"**) to a function called **filterAndRedraw**
- Now we can see why we need global variables!
  - Remember, an **event handler** function only receives one parameter, and it's one that it automatically generated, not one that you specify!
    - All you can specify is the variable that will "catch" this parameter. We have here called it **event**.
    - You may end up not even using this variable, except as a reminder that you've written an event handling function
    - In the code for **filterAndRedraw**, we show another way of addressing the object that triggered the event (instead of using **document.getElementById()**).
      - This can be especially handy if you have multiple document elements triggering the same function!
  - So, any data that we want **filterAndRedraw()** to work with needs to be **global**
    - There are more sophisticated ways of dealing with this situation, but they're beyond the scope of this course. Global variables are a beginner-friendly way of working that is fine for this course!

What's going on with **filterAndRedraw()**?

- Right now, if you play with the popup menu:
  - Selecting anything other than **"All of 2015"** results in the scatterplot disappearing.
  - Further, even if you switch back to "All of 2015", the dots are still gone!
- Notice that the X and Y axes remain even though the dots disappear!
  - Recall how we broke our drawing into a couple of separate functions
  - The **clearPoints**() function that gets called here removes all of the dots, using syntax that you learned when we created the heatmap.
  - However, the **plotPoints()** function doesn't seem to be drawing anything, even though we know it worked initially!
- Do some detective work to figure out what's wrong
  - You may be able to figure this out just by reading this section of code
  - You also should use the usual tools to check for errors, confirm variable values, etc.

Review Challenge #2: **filterAndRedraw()**

- The problem right now is that this call to **plotPoints()** is sending an empty array **filtered** to provide the data for drawing the dots
- Write code that says:
  - If the user chose month **"all"**, then **filtered** should have the same value as **data**
  - Otherwise, **filter** the data so that **filtered** only contains data for the selected month
  - Reminder of *Array*.**filter()** helper function:

```
function (value) {
    if (condition to include item) {
        return true
    }
    return false
}
```

- QUESTION: Why do we need to declare **filtered** outside of the conditional logic that we're writing to determine its value?

Review Challenge #2: Solution

- Here one possible version of the the conditional logic:

```
if (monthChoice == "all") {
    filtered = data
} else {
    filtered = data.filter(function (value) {
        return (value.month == monthChoice)
    })

}
```

- Note that we have reduced the conditional logic for our filter down to a single line.
  - Your filter function must return **true** or **false**, but it can do this any way that you know how
  - In this case, we wrote a **Boolean statement**: the same kind of thing that you write inside the parentheses for an **if()**, but it just provides the **true** or **false** directly
  - You can write the more beginner-friendly **if/else** version if that makes more sense to you!

Review Challenge #3: Connect the Dots!

- It's a bit tough to see the connection between low and high temperatures for each city.
- To practice working with data, and to create an actual simple Cleveland Dot Plot, let's connect the dots for each city:
  - For each item of **data**, create a **line** of class **stems**
    - Be sure to not select other lines in the visualization; you'll get weird results!
  - Draw each line so that it connects the **low** and **high** temperature for that data
  - Position the line on the Y axis so that it lines up with the **city** that it represents
  - Ensure that the lines end up behind the dots instead of in front of them
- REMINDER: the **"line"** object has 5 required attributes: **x1, x2, y1, y2, stroke**
- HINT: All of the other information that you need can be found in the code for drawing the other parts of this visualization
- TIP: To see your work more clearly, you may want to choose a month from the popup menu that we just made functional, so you only see 1 pair of dots for each city!

Review Challenge #3: Solution

```
drawing.selectAll("line.stems")
    .data(data)
    .join("line")
    .classed("stems", true)
    .attr("x1", function (value) {
        return xScale(value.temps[0])
    })
    .attr("x2", function (value) {
        return xScale(value.temps[1])
    })
    .attr("y1", function (value) {
        return yScale(value.city)
    })
    .attr("y2", function (value) {
        return yScale(value.city)
    })
    .attr("stroke", "grey")
    .attr("stroke-width", "1px")
```

- To draw the lines **behind** the circles, they need to be drawn **before** the circles in your code
- Did you remember to use the pattern of selecting a **class** as well as the **type** of shape you wanted?
- To remind yourself of the importance of this, try changing **line.stems** to just **line**. You should see some issues!

*As a **bonus challenge for later**, consider extending and restyling the ticks for each city to extend all the way across the X axis to form dashed graph lines, potentially improving the readability for each city's row of data.*

Customizing Cleveland Dot Plot

- Complete solution can be downloaded as **cdp_framework_completed.zip**
  - We recommend that you spend time trying to understand how to create that version instead of just using the finished product
    - A term project that just slightly customizes this code will not earn a good score! You need to really make the project your own.
    - A good way to do this is to build up the Cleveland Dot Plot from the basics that you know, rather than just following our recipe
- The only really new thing here is the addition of an interactive component, and we even want to see customization in that.
  - Consider if something other than the popup menu in our example would be best for your data.
    - Maybe checkboxes? Radio buttons? It really depends on what you're trying to modify!

Priorities are clarity and creative coding!

- We should be able to easily compare the data that helps to illustrate your hypothesis
  - (whether it ends up being true or untrue; we just need to be able to draw this conclusion based on what we see!)
- Find little ways to enhance your project to show your data as clearly as possible.
  - For example, wouldn't it look better if for the **All of 2015** option we only showed the lowest and highest temperature for each city for the year instead of all temperatures?
    - This is something that you can do with what you've learned in this class. The challenge is just to put the pieces together in a way that works with your data!
  - Another idea: what if we wanted to show temperatures for every **month** on separate lines, so each **city** would be a band, but each **month** would be a point within the band?
    - For this particular visualization, you'd end up with 120 pieces of data in your Y axis, which would be quite cluttered, so this might not be a good idea.
    - However, this is again something that you can do by combining things that you've learned in this class. You just need to figure out how to develop each part, step-by-step!
      - Make sure one thing works before you move on to the next thing and you'll 100% understand your project in the end!