11.0.1: Creating Dynamic Content

In this module, we begin our JavaScript journey by looking into its technical aspects and how it applies to the field of data analytics and visualization. Watch the following video to start your dive into the world of JavaScript.

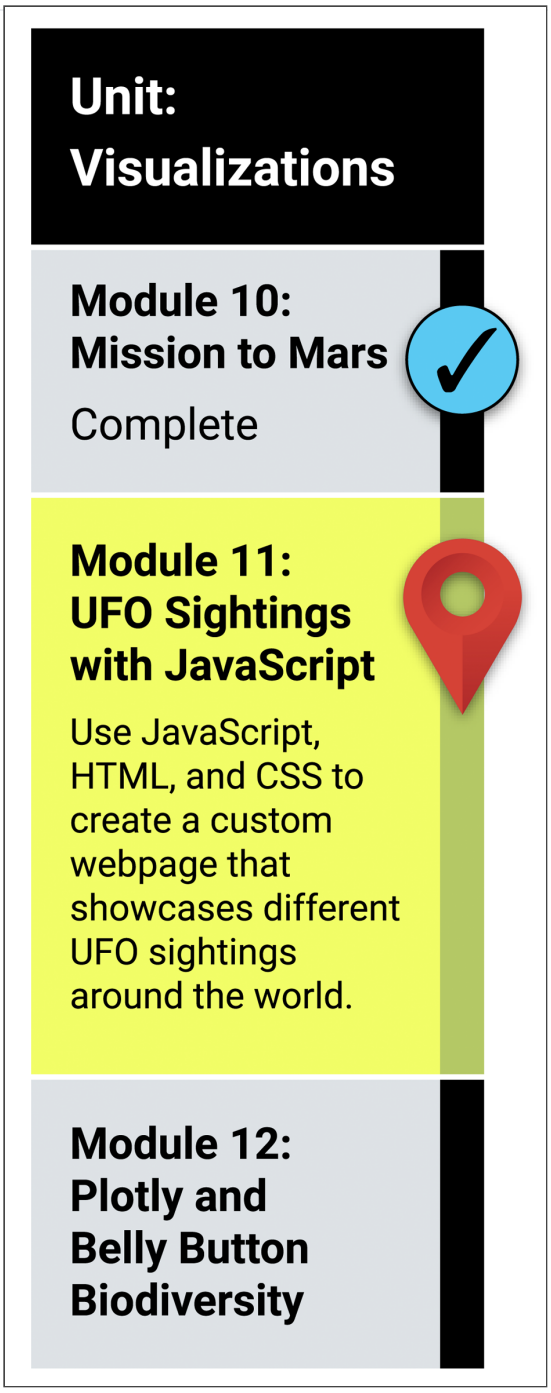
11.0.2: Module 11 Roadmap

Looking Ahead

In this module, you will create a table to organize UFO data that is stored as a JavaScript array, or list. This table will have the ability to filter data based on certain criteria and will be created using JavaScript as the primary coding language.

JavaScript is a well-established coding language that was designed to enhance HTML. It's the backbone of many popular visualization libraries, such as Plotly, and is often used to create custom dashboards. JavaScript also provides a high level of customization: the dashboards built to deliver visual data, such as maps or graphs, can be as simple or complex as needed.

To be successful in this module, you'll need to apply your HTML and Bootstrap skills and have an open mind regarding semicolons.



What You Will Learn

By the end of this module, you will be able to:

- Explain the strengths and weaknesses of JavaScript “standard” and JavaScript version ES6+.
- Describe JavaScript syntax and ideal use cases.
- Build and deploy JavaScript functions, including built-in functions.
- Convert JavaScript functions to arrow functions.
- Build and deploy `forEach` (JavaScript `for` loop).
- Create, populate, and dynamically filter a table using JavaScript and HTML.

Planning Your Schedule

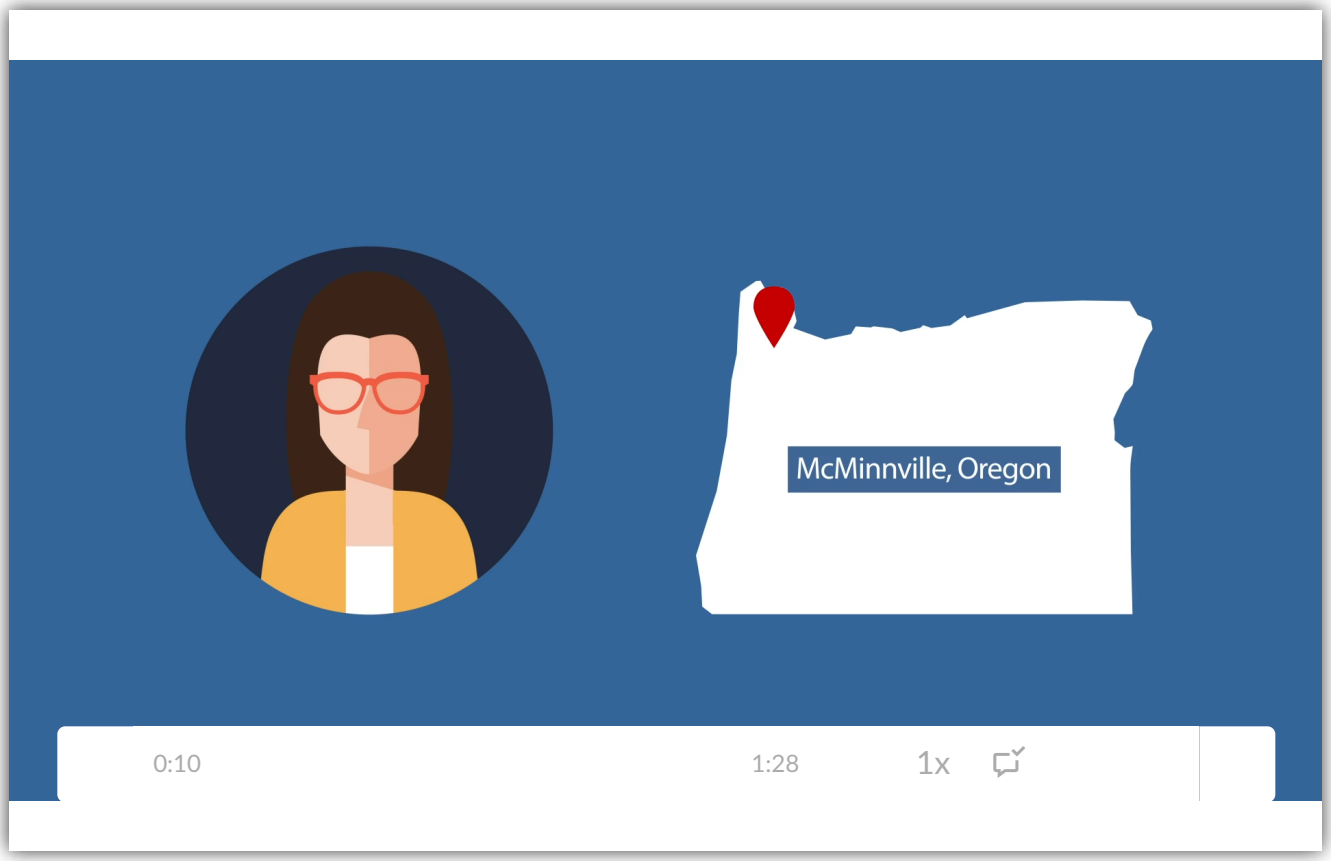
Here’s a quick look at the lessons and assignments you’ll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to Module 11(15 mins)
- JavaScript Basics (1 hour)
- Building Webpages with JavaScript (1 hour)
- Functional JavaScript (1 hour)
- JavaScript for Loops (1 hour)
- Building Dynamic Tables (1 hour)
- Build the HTML (2 hours)
- Application (5 Hours)

11.0.3: JavaScript, Bootstrap, and UFOs

In this module, you'll build a table using data stored in a JavaScript array. You'll also create filters to make this table fully dynamic, meaning that it will react to user input, and then place the table into an HTML file for easy viewing.

You'll customize your webpage using Bootstrap, and equip your table with several fully functional filters that will allow users to interact with our visualizations. Watch the video to learn more about the specific data project you'll be working on.



Create Your Repo

Create a new repository for this module named “UFOs” and clone the empty repo into your class folder.

11.1.1: Overview of ES6+

Dana rolls up her sleeves to get to work. Her first step is to learn a bit about JavaScript: What are the functions of this language? What versions are out there? The first thing she discovers is that JavaScript was developed back in 1995, which means it's certainly proved its worth, since it's still so ubiquitous today! Dana also finds out that there was a major update to JavaScript not too long ago, called ECMAScript. She knows that updates are part of every major language—they fix bugs, provide efficiencies, and generally improve things for developers, so she's delighted that there was such a significant update so recently.

Now, dive in and see just what this update does for JavaScript—and for Dana.

ECMAScript. It sounds as alien as the UFOs Dana is researching. She has heard of JavaScript before—it's been around forever—so what is this new term that she's uncovered?

ECMAScript, also referred to as "ES," is a scripting language designed to help standardize JavaScript. This means that ES provides guidelines and rules for JavaScript to follow, such as how a function should be created to run correctly, also known as the **proper syntax**.

Because ES has provided standardization for JavaScript, it also brings updates to the language. There are updates to every major coding language that fix bugs, update code, and provide overall quality of life improvements for the developers. ES6 is no exception!

There have been many updates to ES over the years, though the sixth update was a major one. You'll probably see "ES6+" mentioned out in the wild pretty often; this is a reference to the "big" update (ES6) as well as the later ones. It's also commonly known as "ES2015" or "ECMAScript 2015." (It was such an important update that it's even known by its year!) There are quite a few different ways to reference this language, but we'll be referring to it as ES6, JavaScript, or JS in this module.

Is ES6 the most recent update?

- No, there have been other updates, though they aren't as large as ES6.
- No, there have been other updates that are equally as large and impactful as ES6.
- Yes, it's the most recent update that brought the highest impact to the language since its release.

Check Answer

Finish ▸

Benefits of the ES6 Update

We've briefly mentioned that the ES6 update was useful, but let's talk a bit more about why it was such a big deal.

Imagine two laptops, one old and one new, side by side. They're similar enough: they're close in size and shape and can complete many of the same tasks, but the newer laptop has an edge. It is faster and can perform tasks with greater efficiency than the older model.

JavaScript after the ES6 update is like the newer computer. This update included many updates to the syntax, which streamlined the code and made it easier to both read and write. Additional, quality of life improvements were implemented as well, such as adding Python-like generators and `for...of` loops. Even functions were updated and streamlined!

> **Note**
>
> `for...of` loops is a new syntax associated with JavaScript, so it's okay to not be familiar with it yet! We'll discuss this syntax in more detail as we learn more about the language.

Later editions of ECMAScript brought about new additions as well, but they are for more advanced uses of the language. In this module, our focus will be on basic JavaScript and ES6 capabilities such as arrow functions. Both are still used today, and there's a chance you'll come up against older versions of JavaScript during interviews as well.

11.1.2: JavaScript in the Real World

While JavaScript is increasingly used in advanced programming and machine learning settings, Dana is curious about it because she's heard it can make websites more functional and dynamic, which she definitely wants her webpage to be. Dana envisions a really cool, interactive dashboard, but she'll have to dig into the details to learn more!

JavaScript is one of the powerhouse languages out in the wild today. While its strength is in creating visually appealing and dynamic content, it is starting to grow into other fields as well. Tensorflow, a popular machine learning tool, even has its own JavaScript library now.

It's pretty easy to start feeling daunted by everything JavaScript can do, so Dana is more interested in examples of similar websites—ones that use filters on lots of data.

- **Online shopping websites:** These are a great example of dynamic content. They contain filters for departments, and then filters for items within those departments. Filters on top of filters!
- **Ecological data:** The **National Ecological Observatory Network (NEON) (https://data.neonscience.org/browse-data?showAllDates=true&showAllSites=true&showTheme=org)** has very large and diverse datasets; these are also displayed on their website as dynamic tables with multiple filters.

- **Weather data: The National Snow & Ice Data Center (NSIDC) (https://nsidc.org/data/search/#keywords=permafrost/sortKeys=score,,desc/facetFilters=%257B%257D/pageNumber=1/itemsPerPage=25)** also has very large datasets presented in table format on their website. These tables include filters and parameters that can be applied to their table.

11.1.3: Writing JavaScript

While JavaScript is clearly capable of a variety of tasks, Dana plans to start with something a bit more manageable. Instead of building an entire dashboard right away, first she'll create a filterable table to display the data. She decides to dig into the syntax of the language. It's very different from other languages Dana has encountered before, so she wants to be sure she understands the basics before she begins to build anything.

One major component of each coding language is its **syntax**. For example, Python is a pretty clean and easy-to-read language; there aren't many semicolons, and the indentation and spacing makes sense. SQL, on the other hand, includes semicolons, but it also has guidelines and requirements when it comes to indentation and spacing.

JavaScript is no different: there are guidelines and requirements for writing it. But because JavaScript can be added to an HTML page, there are more guidelines and requirements than for languages that can only live in a `.js` file or Jupyter notebook such as Python. There are a few important things to remember about JavaScript syntax. We'll start with the following:

- Case sensitivity
- Semicolons
- Statements and expressions

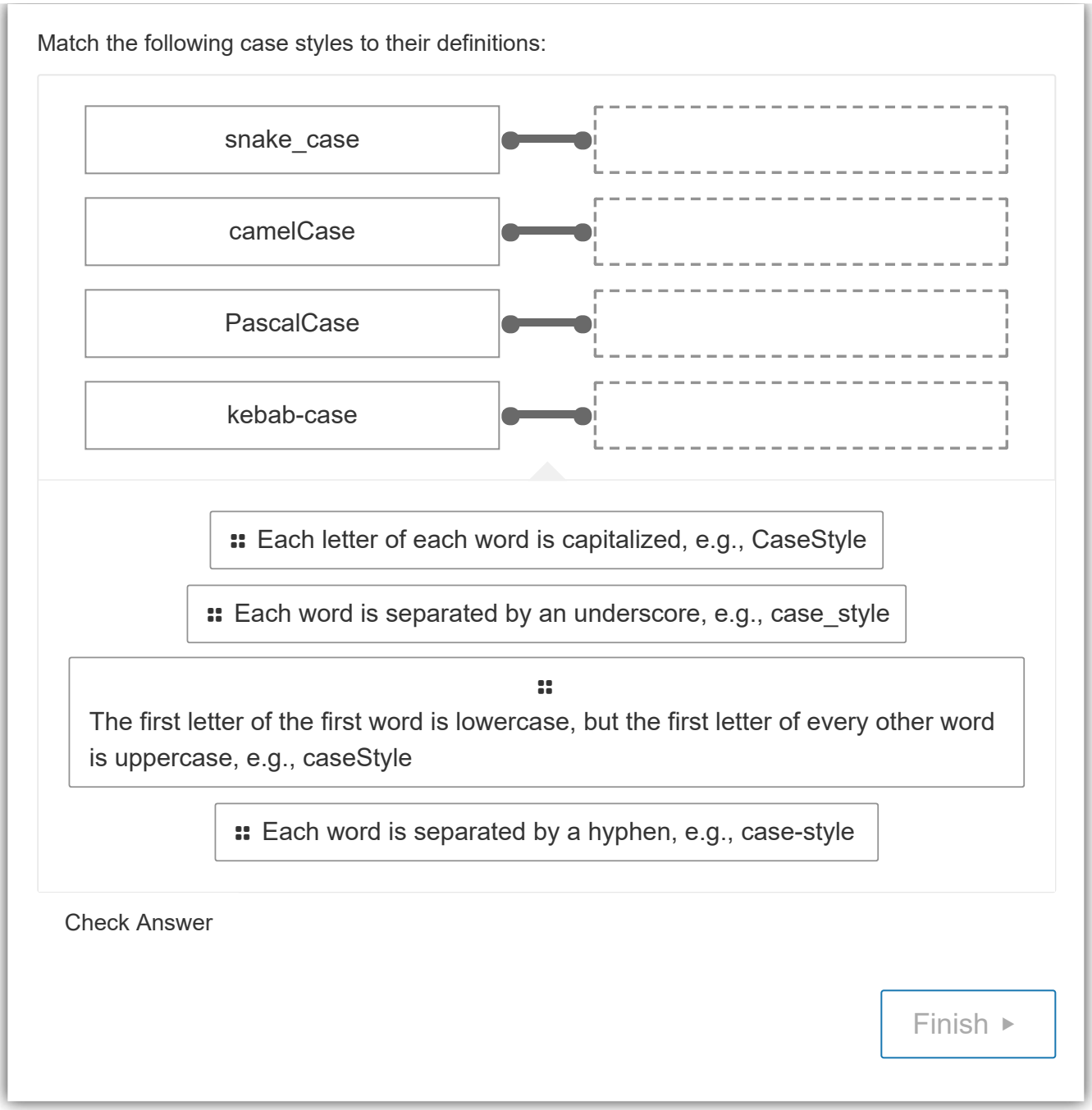
- Code blocks

We'll be sure to get in lots of practice so that Dana can feel 100% confident in her skills.

Case Sensitivity

JavaScript is case sensitive. **Case sensitivity** means that JavaScript considers upper- and lower-case words to be different. For example, if we were to assign the words "data" and "Data" as variables, we would be able to save different information in each word. Of course, actually doing this with the word "data" could lead to confusion pretty quickly. Instead, just remember JavaScript cares about capital letters.

Similarly, JavaScript uses different naming conventions than Python that involve case sensitivity. Different languages utilize different methods to link words without using spaces, which is called a **case style**.



JavaScript's code style, according to coding guidelines and syntax, is camel case. You'll encounter this case often as you begin to practice your coding. It's especially useful when declaring variables.

> **Note**
> Camel case is the preferred naming convention in JavaScript. This is especially helpful in cases where Python data is used. For example, we would know that variables named with snake case originated from the Python side of things.

Semicolons

Much like SQL, when coding in JavaScript it's good practice to end statements with a semicolon. Technically, they are optional when it comes to executing your code, but they are helpful because they tell JavaScript that a particular line or block of code is complete. It's considered a best practice to include semicolons throughout your code. You'll encounter many semicolons throughout this module.

Let's use a print statement as an example. In JavaScript, a print statement is called a **console log**. To print "Hello, world!" to the console, we would use this line:

```
// Printing a string with JavaScript
console.log("Hello, world!");
```

> **Note**
> While the `print()` function does exist in JavaScript, it will actually try to print to a printer instead of our console.

This statement is almost identical to a basic Python print statement, as shown below.

```
# Printing a string with Python
print("Hello, world!")
```

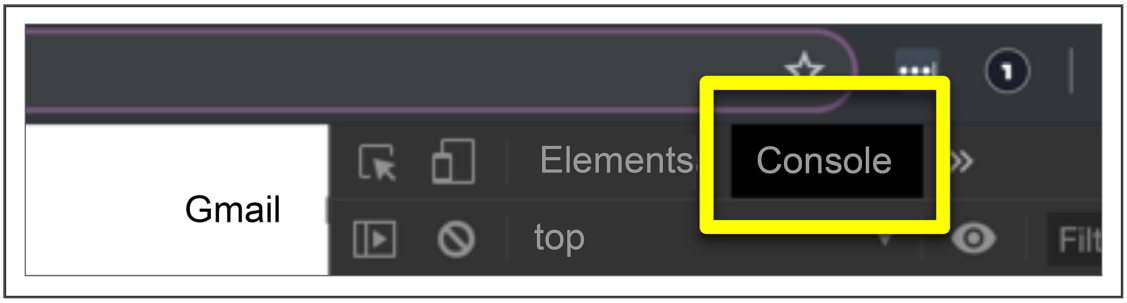
Both methods will print the string (in this case, "Hello, world!"). But in addition to switching "print" with "console.log," in JavaScript, a semicolon has been added at the end of the statement.

Testing Simple Statements

Simple JavaScript statements such as `console.log()` can be tested using DevTools. For example, follow these steps to test `console.log("Hello, world!")`.

1. Go to a site like **Google** **(http://www.google.com)** and activate your DevTools. This is where we'll access our console; the console is the command line interface tool we'll use to test JavaScript, much like our terminal is used to test Python.

   You can use any site to open your DevTools; it isn't a requirement to use the Google search page.
2. Click the "Console" tab at the top of the screen.



3. Type `console.log("Hello, world!");` on the first line and then press Enter.

The Console tab in DevTools will become a very important tool when we begin to code later on. The Console tab will allow us to see if an error has occurred and, if so, which line of code is causing the disruption.

Why will we be using DevTools to check and debug our code?

- We'll use DevTools because using VS Code to run and debug JavaScript code takes longer and doesn't look as neat.
- We'll use DevTools because DevTools will keep code in memory if we refresh the page.
- We'll use DevTools because the console available in DevTools is our JS command line interface.

Check Answer

Finish ▸

Statements and Expressions

When describing JavaScript code, the terms "statements" and "expressions" are both used, and often. Here's how to distinguish between the two:

- Statements perform actions.
- Expressions create values.

Assigning a variable is an example of a statement. Using arithmetic to create a new value is an expression.

Code Blocks

Code blocks, which we will see more often as we start writing functions, are denoted by curly brackets. Code inside the curly brackets are typically indented two to four spaces. This isn't required to run the code, but it does make reading it easier and follows the coding guidelines.

11.2.1: JavaScript Components

Now that Dana knows where she'll encounter JavaScript in the wild—as well as some of the key differences between JavaScript and Python—it's time to get down to business. Understanding the basics of a programming language from text is one thing, but putting that understanding into practice is another. Dana is ready to dive in and start working with some of the basic components: variables and lists.

We're going to begin our practice by familiarizing ourselves with some basic JavaScript components: **variables** and **arrays**.

Variables

Before ES6 came along, there was a single way to declare a variable: `var`. You've already worked with variables in Python, but this concept in JavaScript is a bit different. Let's compare a Python variable to a JavaScript variable:

Python	JavaScript
y = 2	`var` y = 2;

Python's way of assigning a variable is quite simple: type the name of the variable followed by its value: `y = 2`.

JavaScript is similar, but with two additions: add `var` before the variable, and then add a semicolon after the value, like this: `var y = 2;`.

What are the main differences between assigning a variable in Python and JavaScript?

- ○ There really aren't any differences between the two. Variables are assigned the same way in both languages.
- ○ In JavaScript, a variable is assigned by first declaring it. Completion of the statement is indicated with a semicolon.
- ○ JavaScript is a little more lax with the syntax; declaring the variable is more explicit, but only until developers are used to the language. Then variables are declared the same way in both languages.

Check Answer

Finish ▸

Let's test our JavaScript variable assignment using DevTools. If you still have your console open and ready to go, great! If not, go ahead and bring another up to practice with.

Skill Drill

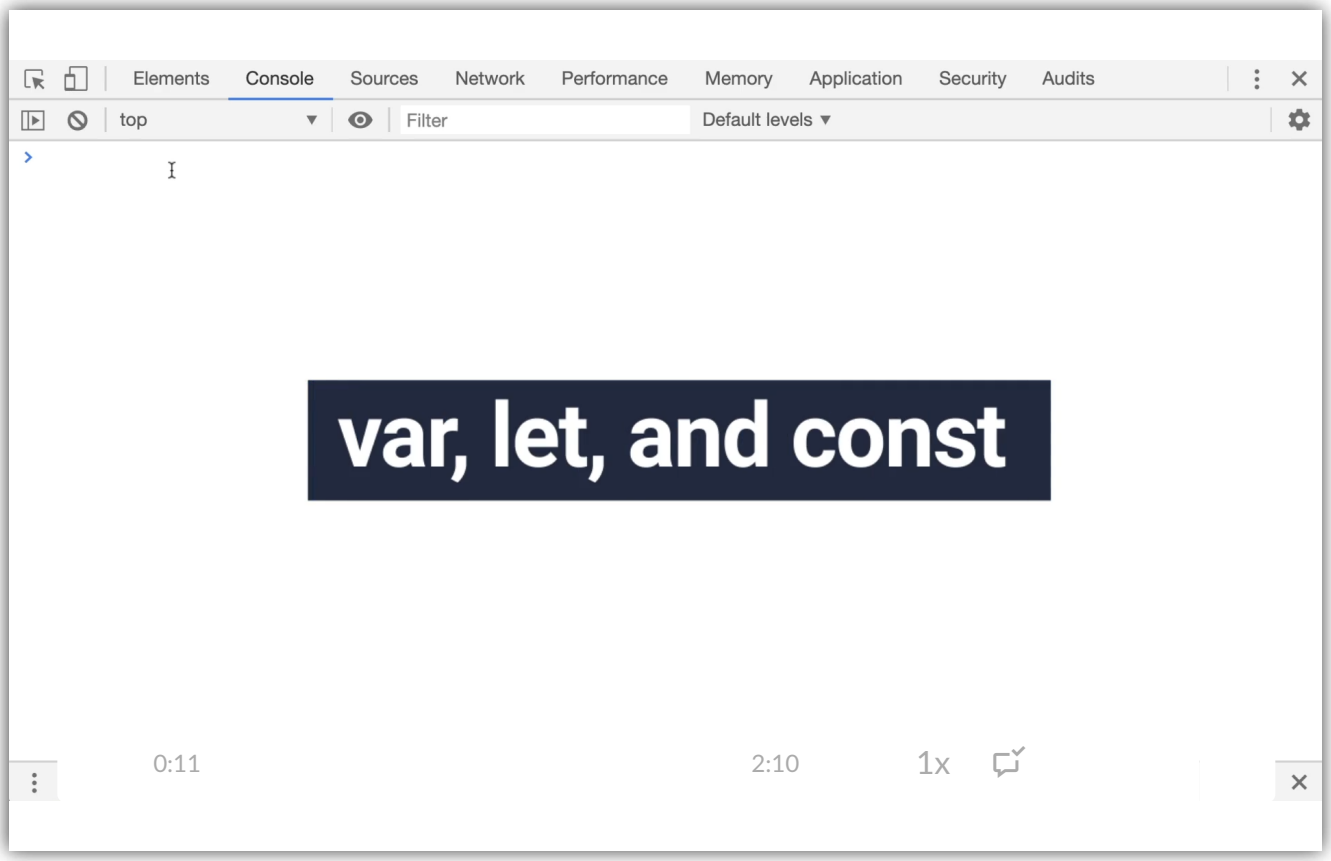
If needed, visit a webpage such as **Google (http://www.google.com)** and activate your DevTools. Click the Console tab to activate the console. Then do the following:

1. On the first line, type `var y = 2;` and press Enter.
2. On the next line, type `console.log(y);` and press Enter.

The value of `y` should print to your console.

Of the many additions that came along with ES6, two more ways to declare variables were also introduced: `let` and `const`.

This can be a bit trickier than it seems, because in JavaScript a variable isn't always just a variable. There are specific uses for different variables, and using `let` and `const` instead of `var` helps developers define what the uses are. Let's check them out in more detail.



Create Variables with let

The biggest difference between `var` and `let` is that the `var` declaration is global, meaning it applies to the program instead of being contained in a block of code.

When a developer chooses to use `let`, it basically means "I might want to use this variable again later to hold different data, but in this code block I'll only use it once." In ES6+, `let` is typically used in place of `var`. We'll be using `let` in this module, but both are encountered out in the wild.

Create Variables with const

The `const` declaration is more specific than `let`. Instead of being contained within a block of code, `const` tells JavaScript that the variable won't be reassigned or redeclared, either in a block of code or within the program as a whole. The following table highlights the key differences of `var`, `let`, and `const`:

Least specific	**var**	**Variable used in entire program**
	let	Variable used in a code block
Most specific	const	Variable used once

Now that we've discovered three different ways to declare variables in JavaScript, let's take a look at arrays.

Arrays

When coding in Python, data can be grouped together in a **list**. The same is true of JavaScript. In fact, Dana was inspired to learn JavaScript because the data is already stored in a JavaScript array! Let's take a look at the data to see what we're working with. Start by downloading the JavaScript file below:

Download data.js
(https://courses.bootcampspot.com/courses/138/files/14467/download?wrap=1)

> ### Adding to your Repo
> This `data.js` file is a large part of this project. Save it in the new repository you cloned to your computer.

When you open the file, you'll see that it contains *a lot* of data. Look at the construction of this array.

First, the name of the array, `data`, is declared with `var`:

```
var data = [
```

The structure of the array begins much like a Python list: with a square bracket. But the data inside is arranged a bit differently.

Each entry is inside the square brackets, like a Python list. That's where the similarities begin to taper off.

In this particular JavaScript array, we're not recording a single item and moving on to the next, much like a simple list (such as `[1, 2, 3]`). However, here we're recording an entire event: date, location, type, and even comments are saved inside a single array. Not only that, but multiple events are recorded. Because we have so much information, the array looks more like a Python dictionary than a simple Python list. Take a look at the example below:

```
  {
    datetime: "1/1/2010",
    city: "benton",
    state: "ar",
    country: "us",
    shape: "circle",
    durationMinutes: "5 mins.",
    comments: "4 bright green circles high in the sky going in circles th
  },
```

Within a set of curly brackets, we can see the key-value pairs such as the date, city, and state.

While this looks somewhat similar to a Python dictionary, there is one key difference. Scroll to the last link of the data file and see if you can spot the difference. You can use keyboard shortcuts to reach the bottom quickly, instead of scrolling through the whole file.

What is the difference between this JavaScript array and a Python dictionary?

- The array was closed with a square bracket and a semicolon immediately after (`];`). The semicolon signals that this block of code is complete.
- There isn't a difference; both a Python dictionary and this JavaScript array are completed by adding the other curly bracket.
- There isn't a difference; the JavaScript array is closed with a square bracket, just like a Python list.

Check Answer

Finish ▸

There's a lot going on in this file: for Dana's article to be a success, she will need to make these sightings easier for people to visually parse by converting them from their current state, a JavaScript array, into an HTML table.

How is a JavaScript array like the one in our data.js file different from a Python list? Select all that apply.

- [ ] The completed block of code is signaled by a semicolon.
- [ ] Multiple events are recorded as part of a collection (date, location, time, for example) instead of a simpler collection such as a list of numbers (like 1, 2, 3).
- [ ] Unlike a Python dictionary, the key value in a JavaScript object does not need to be in quotes.

Check Answer

Finish ▸

Convert the Array to a Table

To convert the array to a table, we're going to take the following code and turn it into the table shown below:

```
{
  datetime: "1/1/2010",
  city: "benton",
  state: "ar",
  country: "us",
  shape: "circle",
  durationMinutes: "5 mins.",
  comments: "4 bright green circles high in the sky going in circles then one bright green light at my front door."
},
{
  datetime: "1/1/2010",
  city: "bonita",
  state: "ca",
  country: "us",
  shape: "light",
  durationMinutes: "13 minutes",
  comments: "Three bright red lights witnessed floating stationary over San Diego New Years Day 2010"
},
```

Date	City	State	Country	Shape	Duration	Comments
1/1/2010	benton	ar	us	circle	5 mins	4 bright green circles high in the sky going in circles then one bright green light at my front door.
1/1/2010	bonita	ca	us	light	13 minutes	Three bright red lights witnessed floating stationary over San Diego New Years Day 2010

The first step in transitioning the data from an array to a table is to create the appropriate variables using `var`, `let`, or `const`. Open VS Code and create a file in our repo folder named `app.js`. This is where we'll keep the code that builds the HTML table and fills it with data from `data.js`.

11.2.2: Organize Your Repository

Dana has started to get into a JavaScript coding rhythm. But before she begins to create code for real and then commit scripts to her repo, she needs to organize it.

Building a page that contains JavaScript will require Dana to link additional JavaScript files to the `index.html` file that she'll be working on later. This means keeping track of multiple things at once: an HTML file, JavaScript files, images (for customizing the webpage) and a CSS style sheet. Therefore, it's a good idea for Dana to establish a solid folder structure now instead of when she's elbow deep in creating her JavaScript functions.

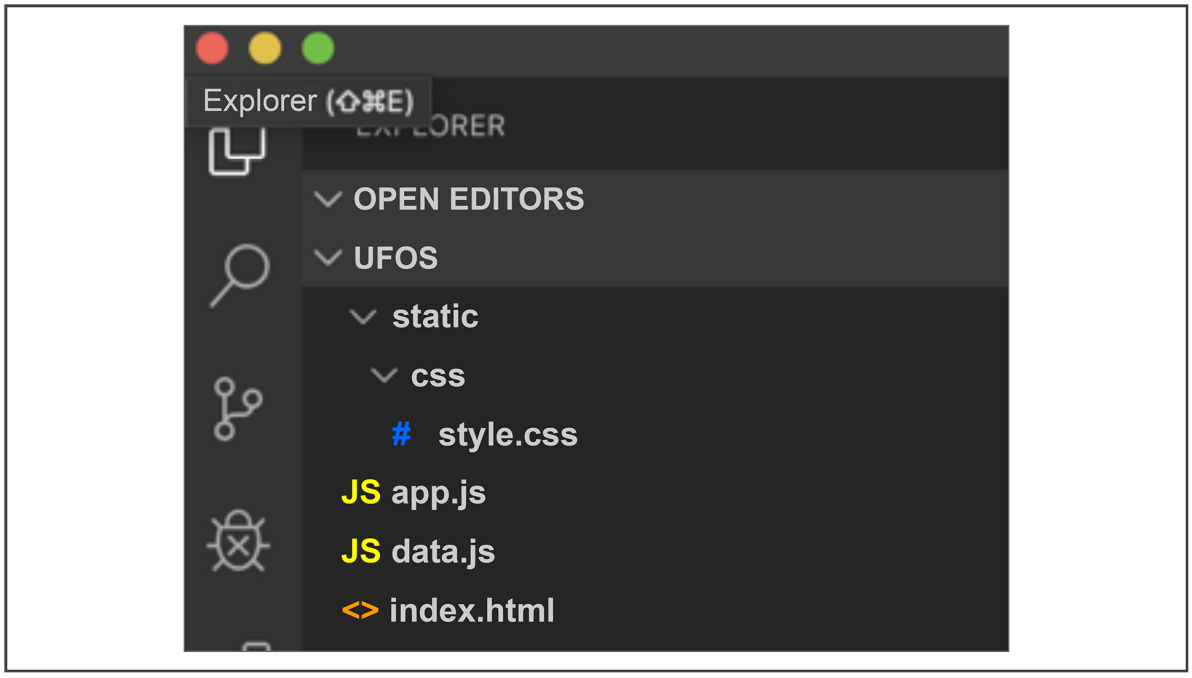
Before we get too far along with our coding, we need to set up a file organization system for our repo. The end result of this project will be an HTML page or application, so we need to establish the proper folder structure accordingly. At a high level, here's what we'll do:

- Create the `index.html` file.
- Create a subfolder to hold the CSS file (style.css).
- Create a subfolder for images.
- Create a subfolder to hold JavaScript.

First, in the repo folder we established earlier ("UFOs"), create the `index.html` file. This file is the window to our work: the table and Dana's article summary (along with titles and filters) will all be displayed through this file. We won't be coding it yet—that will come later—but we're creating it now so that it will be ready for us when it's time to build the page.

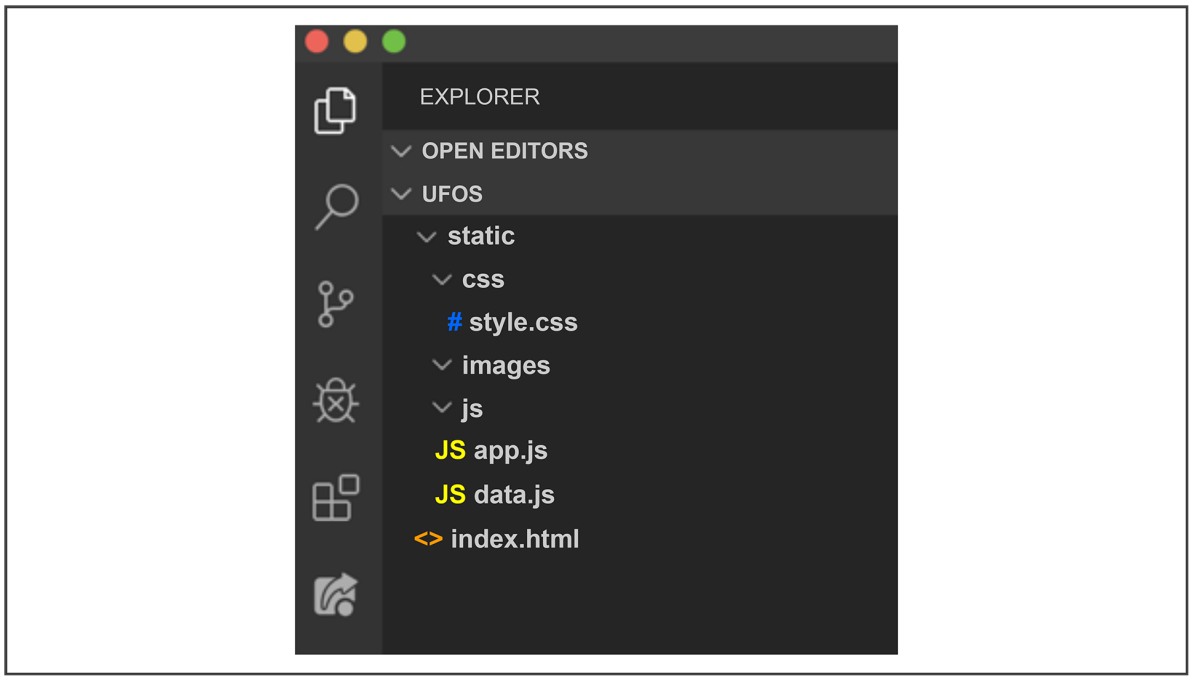
Next, create a subfolder in the repo folder named "static." This static folder will hold our static CSS file; this only means that it isn't being moved or altered externally. In VS code, right-click the menu and select "New Folder," and then name it "static."

Inside the static folder, create another subfolder named "css" to hold the `style.css` file. You can use the same right-click method to bring up the creation menu, but this time select "New File." We'll customize our webpage using the `style.css` sheet, but for now we can leave it blank. Here's what the folder structure should look like so far when viewing it in VS Code.



The next subfolder we'll create is our "static" folder to hold whatever images we want to add to our website when it's time to customize it. Create the folder now and name it "images." But for now, move on to the next step—we'll add images later.

The third and final subfolder we'll create is one to hold our JavaScript. Name the folder "js" and move the `data.js` and `app.js` files into it so that your folder structure looks like this:



Establishing this folder structure is a best practice when creating webpages with JavaScript. It's important to keep things organized when creating a webpage using JavaScript components, as there are even more moving pieces than a static website. We'll be linking to images and a style sheet as well as JavaScript scripts. The organization presented here provides clearly designated spots to store the code we'll be working on, making it easier to locate them as we go.

11.2.3: JavaScript Objects

It's been a good day of research. Dana is far more familiar with some of the basic components of JavaScript: she now knows that variable declaration can actually occur three ways, and the array she is working with is similar to a Python dictionary. It's a great start, but Dana's still a little fuzzy on the array. It looks like a JSON, or a dictionary, so it's more complex than a simple list, right?

Dana's intuition has served her well: the JavaScript array is indeed a bit more than a simple list. Let's take a closer look at JavaScript objects and how to interact with them, which will help us as we begin to create our code.

Coding in JavaScript requires proficiency with JavaScript objects. And, in JavaScript, many different things can be considered an "object." We've actually already encountered one! Let's look at a snippet of code from our `data.js` array:

```
var data = [
  {
    datetime: "1/1/2010",
    city: "benton",
    state: "ar",
```

```
    country: "us",
    shape: "circle",
    durationMinutes: "5 mins.",
    comments: "4 bright green circles high in the sky going in circles th
  },
```

As mentioned earlier, this looks very similar to a Python dictionary or something we'd find in a JSON file. In this code snippet, everything within the curly brackets is considered to be properties of a JavaScript object. The object is our variable: data.

There are several ways we can access the properties, also called key-value pairs or objects, in the array.

Highlight the array in the JavaScript code below.

```
let weatherToday = [
   {
     year: “2019”,
     month: “October”,
     high: “89”,
     low: “65”,
     measurement: “Farenheit”
   }
]
```

Check Answer

Finish ▸

Also, objects are not limited to being contained within an array. In fact, an array itself is an object. Dates are also objects, as are functions; and Booleans *can* be objects. Basically, many things can be—or are—objects. We'll get plenty of practice with objects as we start to build our website.

Before building the website, we should plan it out. By using a storyboard and mapping the elements out beforehand, it will be easier to assemble them later.

11.2.4: Storyboarding

Dana has grown more familiar with JavaScript syntax and her basic code is gaining momentum. She's ready to start putting it to use!

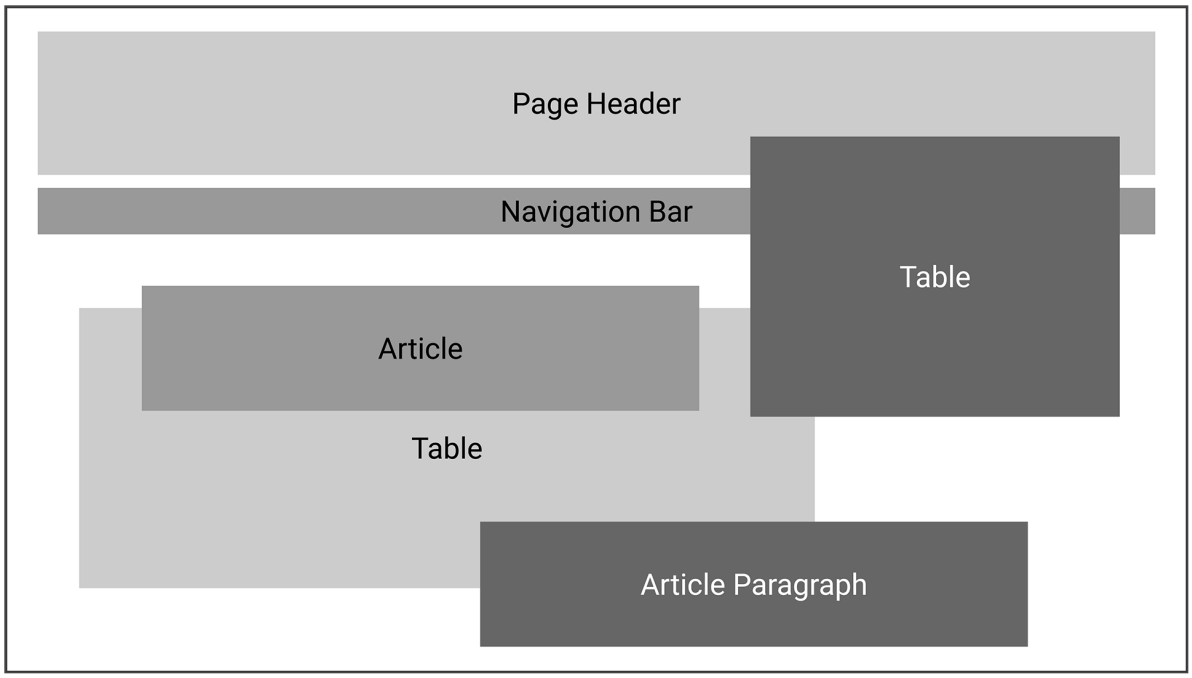
Dana's goal is to create an interactive webpage that allows readers to parse the data around UFO sightings. So, she essentially needs to build two things: the webpage that will allow users to view the data (HTML) and a dynamic table that will present it (JavaScript).

Dana wants to storyboard her website to have an idea of what her readers will see when they view the final product. Storyboarding is incredibly useful in determining the layout of a webpage, so it's important to complete this step early in order to save time later. It's like building a house. You need to know how it's all going to fit together before you start building!

Once the template has been created, Dana can begin to code the JavaScript portion by first importing the data and then referencing it with a variable.

Typically, developers build HTML and JavaScript elements somewhat simultaneously because they complement each other. For example, the JavaScript table will be referenced within the HTML code, and different HTML components will be referenced within the JavaScript code. Because these files are so closely linked, Dana will switch between building the JavaScript table (within the `app.js` file) and the HTML page (within an `index.html` file).

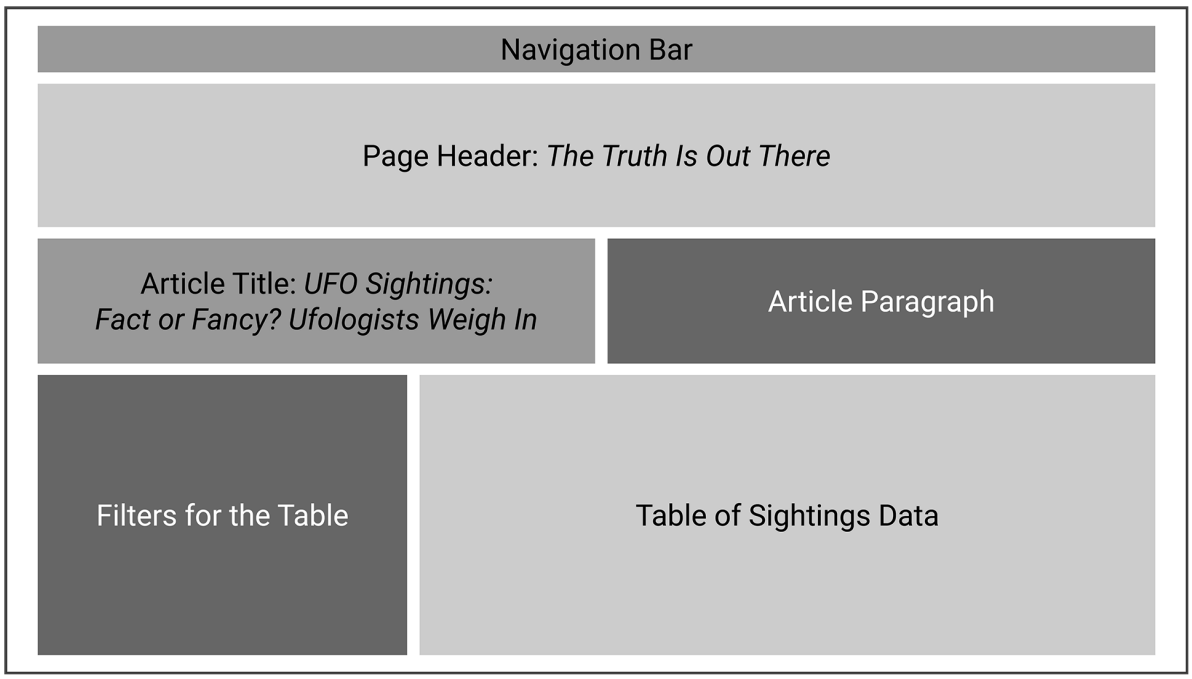
Dana also has a solid idea of how she wants her webpage to look, but it's easy to get lost in the details of building a webpage without a visual reference. A visual reference such as a storyboard will help Dana outline all of the elements she wants included, such as the article title, a summary, and the table itself. Then, when she begins creating JavaScript code to include the table, she'll know exactly which HTML components she'll be connecting to her table. Dana already knows she'll have several individual components on the webpage, shown below:



Now she just needs to figure out how to assemble them. This is where a storyboard comes in.

Create a Storyboard

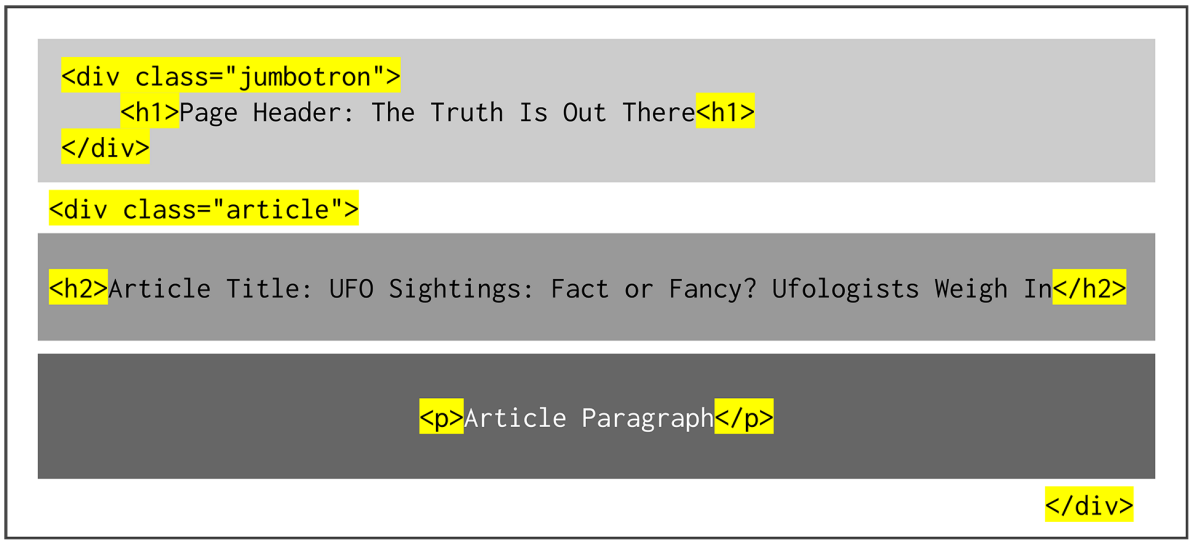
A **storyboard** serves as a kind of blueprint for your site and helps with the transition from idea to finished product. Think of it as a map of the webpage.



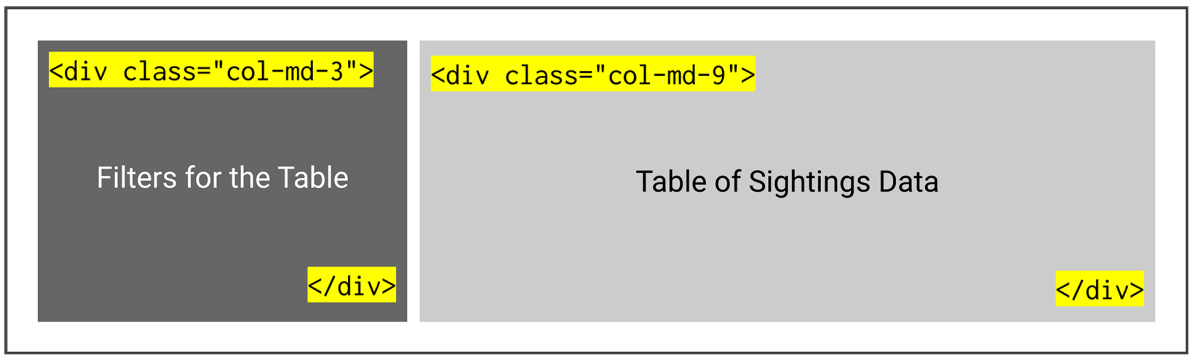
This step is key for a couple of reasons.

First, knowing how we want the webpage to look before building it will save us time later. Second, it helps us make sure we've captured everything we want displayed. Sometimes, seeing the map of the website helps us ensure that all the elements we want displayed are included.

We already know what components we want to use, such as a Jumbotron for the header, and the grid system for the filters and table. See the following image:



We also have an idea of how many columns we want each table component to use.



Now that a storyboard is in place, we can really get going! Let's align our code.

Align the Code

When we align our code, we're putting our plans into action, such as when we start transitioning our storyboard into a webpage. We'll start by building our components. The first one will be the table we generate with JavaScript. Open your `app.js` file with VS Code. The first thing we're going to do is import the data. This won't look like an import from Python. For starters, the double backslash (//) is how you comment your code in JavaScript.

In your code editor, type the following to declare a variable, `tableData`, using `const`.

```
// import the data from data.js
const tableData = data;
```

Why is `const` used to declare the `tableData` variable?

- We don't want the variable to be reassigned or reused at all in our program.
- We want to use the most updated method of declaring variables.
- We want the data to be immutable.

Check Answer

Finish ▸

Next, we need to point our data to our HTML page. Specifically, we need to tell JavaScript what type of element the data will be displayed in. We already know that the data will be displayed in a table, so in our code editor we'll reference the `tbody` HTML tag using D3.

> **IMPORTANT**
> D3 is a JavaScript library that produces sophisticated and highly dynamic graphics in an HTML webpage. It is often used by data professionals to create dashboards, or a collection of visual data (such as graphs and maps), for presentation.

Return to your code editor and type the following:

```
// Reference the HTML table using d3
var tbody = d3.select("tbody");
```

With this code, we:

1. Declare a variable, `tbody`
2. Use `d3.select` to tell JavaScript to look for the `<tbody>` tags in the HTML

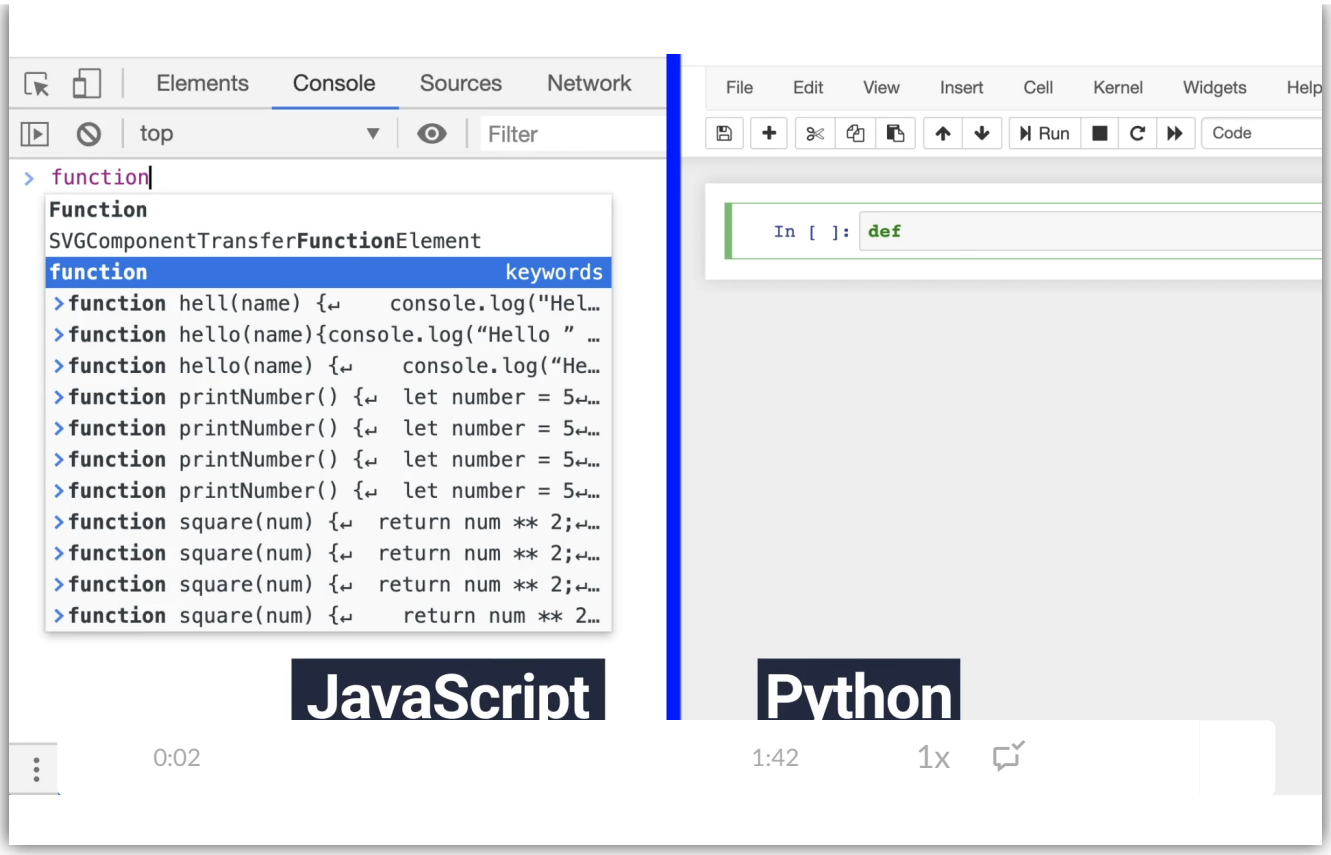
Although we aren't building the HTML right now—we'll do this after we put together the code—we already know that the data will fit into that tag because it's a standard table tag that is used often in HTML, with or without JavaScript enhancements.

11.3.1: Getting Started with JavaScript Functions

Dana has started to build her code, which is really exciting! When she imported the data, she took the first step in building her website. The next step is to build the table to sort and store the data.

Dana knows that building this table will introduce a new level of complexity involving `for` loops and functions. Thankfully, JavaScript and Python have similar logic, so after Dana reviews and practices with code similar to what she'll use in her project, she'll be ready to start integrating it into her code.

Functions in Python and JavaScript have similar logic: we provide the language with a set of instructions to follow, which can then be reused as needed. Watch the following video to learn more about JavaScript functions.



In Python, a simple print statement looks like this:

```
# Simple Python print statement
def print_hello():
    print("Hello there!")
```

In this code, the function is declared with the keyword `def` followed by the name of the function, a set of parentheses, and a colon, with the indented code below.

To write a print statement in JavaScript, we begin the same way: by declaring the function. To do this, we use the keyword `function`. (**Note:** Remember that the JavaScript syntax uses `console.log` instead of `print`.)

```
// Simple JavaScript console.log statement
function printHello();
```

At this point, the process diverges from Python. The next step is to add a set of curly brackets, and then add the indented code between them.

```
// Simple JavaScript console.log statement
function printHello() {
  console.log("Hello there!");
}
```

> **Skill Drill**
>
> Return to the console tab of your DevTools and run the JavaScript function you just reviewed.

Get more practice with Python and JavaScript functions in the following activity.

Choose the correct JavaScript version of this Python function:

```
def show():
    print("Hi!")
```

- ○
```
function show()

    return ”Hi!”;
```
- ○
```
function show() {
    print(”Hi!”)
    };
```
- ○
```
function show() {
    return ”Hi!”;
    };
```

Check Answer

Finish ▸

Let’s take a closer look at basic functions in JavaScript.

11.3.2: Simple JavaScript Functions

Even though Python and JavaScript are logically similar, it's becoming more and more apparent to Dana that they have many syntactical differences. It's a lot to take in, so Dana wants more practice creating and calling functions. She wants to know if arguments are passed in the same way, as well as how functions are called.

Arguments can be passed into both Python and JavaScript functions. Let's take a look at another Python function as an example. Look at the following code:

```
# Takes two numbers and adds them
def addition(a, b):
    return a + b
```

In this function, we've added the ability to input two numbers and add them. Let's convert this same function to JavaScript in the DevTools console. Type the following on a new line in your console:

```
// Takes two numbers and adds them
function addition(a, b) {
  return a + b;
}
```

Using the numbers 4 and 5, how would you run the addition function in the console?

- `log.addition(4, 5);`
- `console.log(addition(4, 5));`
- `console.log(addition(a, b));`

Check Answer

Finish ▸

To test the new function, type `console.log(addition(4, 5));`. This is the equivalent of using a print statement in Python to print the function. Like Python, we can condense the code even further by typing only `addition(4, 5);` to execute the function as well.

> **IMPORTANT**
> In the "addition" function created above, the items within the parentheses are referred to as **parameters**. For example:
>
> ```
> function addition(a, b) {
> return a + b;
> }
> ```
>
> In this function, data points `a` and `b` are the parameters. Think of them as placeholders for the values we will add later, such as 4 and 5.
>
> Functions in JavaScript can have any number of parameters. However, from a practical standpoint, it's not a good idea to have more than two parameters per function. Too many arguments can significantly slow down and even crash your code.

Now practice functions in the following Skill Drill.

Skill Drill

Practice executing the addition function in your console. Try switching up the numbers and printing it with and without the use of `console.log();`.

Functions are a versatile tool in any coding language, and JavaScript is no different. Functions can also call other functions. The code below creates a new function that includes our simple function within it. In your console, type the `doubleAddition` function below:

```
// Functions can call other functions
function doubleAddition(c, d) {
  var total = addition(c, d) * 2;
  return total;
}
```

What does this function within a function do?

- The new function takes two arguments, `c` and `d`, and incorporates the original addition function we wrote earlier to multiply the sum of two numbers by 2.
- The new function just adds numbers twice.
- The new function will multiply the first two numbers given, then take the second function that will also multiply the digits.

Check Answer

Finish ▸

Let's run the new function, `doubleAddition`, with the same figures we used earlier: 4 and 5. Within this function, we're calling our original function (`addition`) and multiplying the sum of 4 and 5 by 2. We've assigned a variable to the function we've already created, so that we can print the total using a return statement.

Run the new function, `doubleAddition`, in your console using the numbers 65 and 34. What is the total returned?

- 188
- 199
- 198
- 201

Check Answer

Finish ▸

So far, we've created a function that performs simple addition and a second function that calls our original function, which is a great introduction to JavaScript functions.

> **Note**
> If the code and output in your console is getting cluttered, type `clear()` and press Enter to clear the working area of your console.
>
> Once cleared, you won't be able to see the code anymore, but you can still access what you've written by using the up arrow key on your keyboard. This allows you to cycle through the different lines of code you've already executed.

Like functions in Python, JavaScript functions are very versatile and can incorporate many other actions, such as incorporating `for` loops, which we will explore shortly.

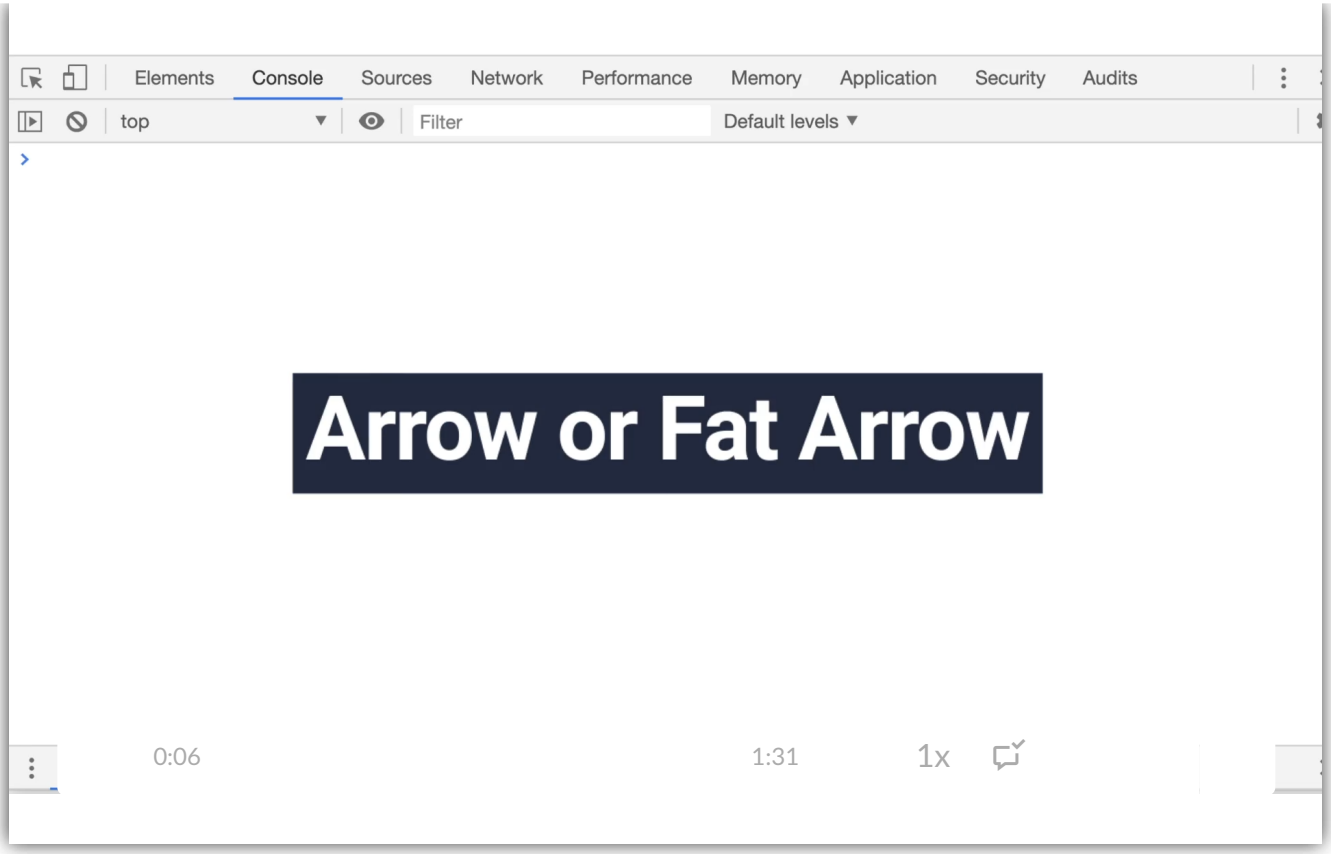
First, let's explore one of the key improvements to JavaScript functions introduced by ES6: arrow functions.

11.3.3: From Simple Functions to Arrow Functions

Having been introduced to JavaScript functions, Dana is now feeling a bit more confident about her JavaScript coding skills. She's excited to explore a shortcut followed by JavaScript insiders: arrow functions, which JavaScript experts use to convert standard functions into a single line of code. That's right: a single line.

Dana is excited about this insider trick because her collection of UFO data is somewhat extensive, and she has a feeling that her code will be complex. She also knows that arrow functions are one of the most popular aspects of the ES6 update, so she's eager to further integrate into the JavaScript community by mastering this function type.

Functions in JavaScript can easily become bulky and difficult to understand. Thankfully, any standard function in JavaScript can be refactored into an arrow function. **Arrow functions** complete the same functions as regular functions, but they use a more compact and concise syntax that makes a code script shorter and easier to read. Watch the following video to learn more about arrow functions.



> **Note**
>
> **Arrow functions** are also known as fat arrow functions because they are introduced with a “fat arrow”: =>
>
> This type of function is very similar to how a Python lambda function is written.

Let’s take a look at a simple function.

```
// Simple JavaScript log statementfunction printHello() {  return "Hello
```

This function, while already fairly short and sweet, can be condensed even further. In the console, type the following code and then press Enter.

```
printHello = () => "Hello there!";
```

When the function is called, our statement will be printed to the console. This is a pretty big change from traditional JavaScript functions.

What are some differences between traditional functions and arrow functions? Select all that apply.

- [ ] There is no return statement.
- [ ] The `function` keyword is missing.
- [ ] `console.log();` isn't included.

Check Answer

Finish ▸

Let's break down the differences in a bit more detail.

1. The arrow function collapses the function from 3 lines to 1 line, which is a significant reduction in characters.
2. The `function` keyword is not part of the arrow function. This is because the arrow symbol (=>) indicates that this block (or line) of code is a function.
3. The `return` keyword and `console.log()` are removed because with this new syntax, JavaScript inherently knows what will be returned.

Let's convert another function, this time with parameters. Here's the original function:

```
// Original addition function
function addition(a, b) {
  return a + b;
}
```

In your code editor, type the following:

```
// Converted to an arrow function
addition = (a, b) => a + b;
```

Once again, a multi-line function has been reduced to a single line. We have removed the `function` keyword, the curly brackets, and the return statement, and added a fat arrow to indicate that "addition" is a function. It's clear and easy to read—and it performs the same way as the original function!

Now let's step it up one more time and convert the `doubleAddition` function, shown below.

```
// Original doubleAddition function
function doubleAddition(c, d) {
  var total = addition(c, d) * 2;
  return total;
}
```

Even this function can be refactored into a single line. Let's begin the process by following the standard syntax: the name of the function, an equals sign, and then the parameters.

```
doubleAddition = (c, d)
```

The next step in refactoring is to add the fat arrow followed by the argument. In this case, the argument is the second function.

```
=> addition(c, d) * 2;
```

> **Skill Drill**
> Use the newly refactored `doubleAddition` function to find the total of 33 and 25.

Familiarity with both types—traditional functions and arrow functions—is important. Both are used often in development, and by the time we're done with this project, we'll have used a combination of the two.

Also, keep in mind that while arrow functions are clear and readable, there are still cases in which traditional functions are necessary. For example, when we want to place a function within another function, we would need to use a traditional function.

11.4.1: Use a JavaScript for Loop

The array containing UFO sightings is huge. Dana knows that iterating through it is inevitable, which means she will definitely need `for` loops. Feeling bolstered by her practice with arrow functions, Dana is ready to up the difficulty a bit by exploring how to create `for` loops in JavaScript.

All coding and scripting languages have a way to iterate through items, such as names in a list. In JavaScript, this process is initiated by the keyword “for” and works in the same manner as a Python `for` loop. Let’s see how this works.

Add the following array to your console.

```
let friends = ["Sarah", "Greg", "Cindy", "Jeff"];
```

Important

Like Python, JavaScript uses zero-based indexing. This means that the first item in an array will be assigned an index placement of 0.

As soon as you press Enter, the words “undefined” will appear directly below your line of code. This is how you know that you’ve successfully executed the line of code and the array has been saved locally.

> **Note**
> Code executed through the console is saved locally, within your system's memory. If you close your console and reopen it, the code will have been erased and you'll need to start over.

To iterate through each name in JavaScript, we can create a `for` loop. First, type the following in your console:

```
function listLoop(userList) {
   for (var i = 0; i < userList.length; i++) {
     console.log(userList[i]);
   }
}
```

Wow, this `for` loop is pretty involved! Let's compare it to a Python loop and examine the differences.

> | JavaScript `for` loop | Python `for` loop |
> | --- | --- |
> | `for (var i = 0; i < userList.length; i++) { console.log(userList[i]); }` | `for i in user_list: print(i)` |

These two loops do the same thing: they iterate through a list and then print each item within it individually. The `for` keyword at the beginning of the loop is the biggest similarity here, so let's start there.

The keyword `for` is the trigger that indicates we'll be iterating through a list. The next line in the JavaScript loop does a few different things, though. This one line can be broken down into three sections.

The following actions occur in this one line:

1. `var i = 0` We assign an iterable variable and set its value to zero. In this loop, think of the letter 'i' to mean 'iterate.' When we assign a zero value, we're starting a counter from the beginning. You can also think of it in terms of list comprehension--the first name of the list has an index value of zero, for example.
2. `i < userList.length`; Here we're basically saying, "If this iterable (`i`) is still smaller than the total number of iterables in the list (`userList`), then move on to the next step."

   So if we're on the second name, but the list is four names long, the `for` loop will continue to loop through it.
3. The final step, `i++`, increases the iterable by 1. We're using list comprehension here; the for loop knows to iterate to the next name because the index number has increased by 1.

When the length of `i` is equal to the total number of items in the list, the `for` loop will complete its iterations and the next line of code will be executed. For example, Jeff's index position is 3; when `i` is equal to 3, the loop is complete. This is because there are no names after Jeff's, nothing with an index value of 4.

What is the result of running the `listLoop` function in your console? **Note:** Remember to add your array of names first.

- ○ Each name is logged in the console, one after the other.
- ○ Nothing, there is a NaN response.
- ○ The names are all printed in the console, but they're still in an array.

Check Answer

Finish ▸

Since our code says to log, or print, each iteration, the names in the array are printed to the console one at a time.

```
> listLoop(friends);
  Sarah                                    VM4077:3
  Greg                                     VM4077:3
  Cindy                                    VM4077:3
  Jeff                                     VM4077:3
```

11.4.2: Practice Using for Loops in JavaScript

It took a bit of work to build the code that iterates through an array, so Dana will create a few more to practice the `for` loop syntax.

First, let's create a `for` loop to iterate through an array of vegetables. Here's our array:

```
let vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"];
```

Now we're going to build the `for` loop. The syntax is exactly the same as it was earlier.

```
for (var i = 0; i < vegetables.length; i++) {
}
```

We're using the keyword `for` to initiate the loop. We also start the loop at the beginning by assigning an iterable as zero with `var i = 0;`.

Next, we tell the loop to continue working through the array as long as the iterable ("i") is less than the number of vegetables in our array: `i < vegetables.length;`.

Finally, we increase our iterable by 1 by adding `i++`; which tells JavaScript to move to the next item in the array until there are no more items.

Let's say we also want each item in the array to be printed to the console. To do this, we'll add a `console.log` statement inside the curly brackets. Let's add a

message to go with each item, too, so it will read “I love [vegetable]” with each iteration.

The final code looks like this:

```
let vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"];
```

```
for (var i = 0; i < vegetables.length; i++) {
    console.log("I love " + vegetables[i]);
}
```

Let’s practice with one more. This time we’ll loop through numbers without using an array.

```
for (var i = 0; i < 5; i++) {
   console.log("I am " + i);
}
```

The only difference between this loop and the previous one is that we aren’t referring to an array. Instead, we are explicitly telling JavaScript to count up to a fifth value.

Highlight the section of code that determines the number of iterations.

```
for (var i = 0; i < 5; i++) {
   console.log("I am " + i);
}
```

Check Answer

Finish ▸

11.5.1: Introduction to Dynamic Tables

Dana is making some headway. She's more familiar with objects and arrays, and she created a few functions, both traditional JavaScript functions and the faster arrow functions. Comparing and writing `for` loops was a bit challenging; logically, they work in the same manner as a Python `for` loop, but the code to create one in JavaScript is far more involved! Dana is realizing that becoming proficient in JavaScript requires a lot of practice and patience.

Practice is progress, though, and Dana is now ready to create her table.

Dana's code is somewhat modest right now, but it's about to get a lot more interesting. Now we're going to help her build the table to display all of the UFO sightings. We'll need to iterate through the array of objects in our data file and then append them to a table row. All of this will happen within a function, which makes the code self-contained.

Creating self-contained code makes it easier to reuse the code and keeps us organized: the code in this function will be used to fill the table with data **only**.

Let's get started by returning to our `app.js` file in the editor and, on a new line, creating a new function.

Typically, functions are named after what they do. We're building a table, so we'll name the function "buildTable." We'll also pass in "data" as the argument. Remember that we used the variable "data" earlier to import our array of UFO sightings? This is the first step in actually working with the data.

In our editor, we should have the start of a new function:

```
function buildTable(data) {

}
```

We're using a standard JavaScript function instead of an arrow function because of what we'll be inserting inside the function (hint: another function!). Let's start building out the rest of the function.

In the next line, we'll want to use code to clear existing data.

Why should you clear existing data from the table?

- You don't need to, but clearing the existing data follows coding guidelines.
- It's a good idea because it keeps the code nice and tidy.
- Because we need to clear the data first, otherwise the data users search will already be filtered when they search again.

Check Answer

Finish ▸

Clearing the existing data creates a fresh table in which we can insert data. If we didn't clear existing data first, then we would find ourselves reinserting data that already exists, thus creating duplicates and making a bit of a mess. It's good practice to clear the existing data first to give ourselves a clean slate to work with.

The line we'll use to clear the data is `tbody.html("");`. But how exactly is this code clearing data?

- `tbody.html` references the table, pointing JavaScript directly to the table in the HTML page we're going to build.
- The parentheses with empty quotes (`("");`) is an empty string.

Basically, this entire line—`tbody.html("");`—tells JavaScript to use an empty string when creating the table; in other words, create a blank canvas. This is a standard way to clear data.

Here is what our code looks like with the addition of this line:

```
function buildTable(data) {
  tbody.html("");
}
```

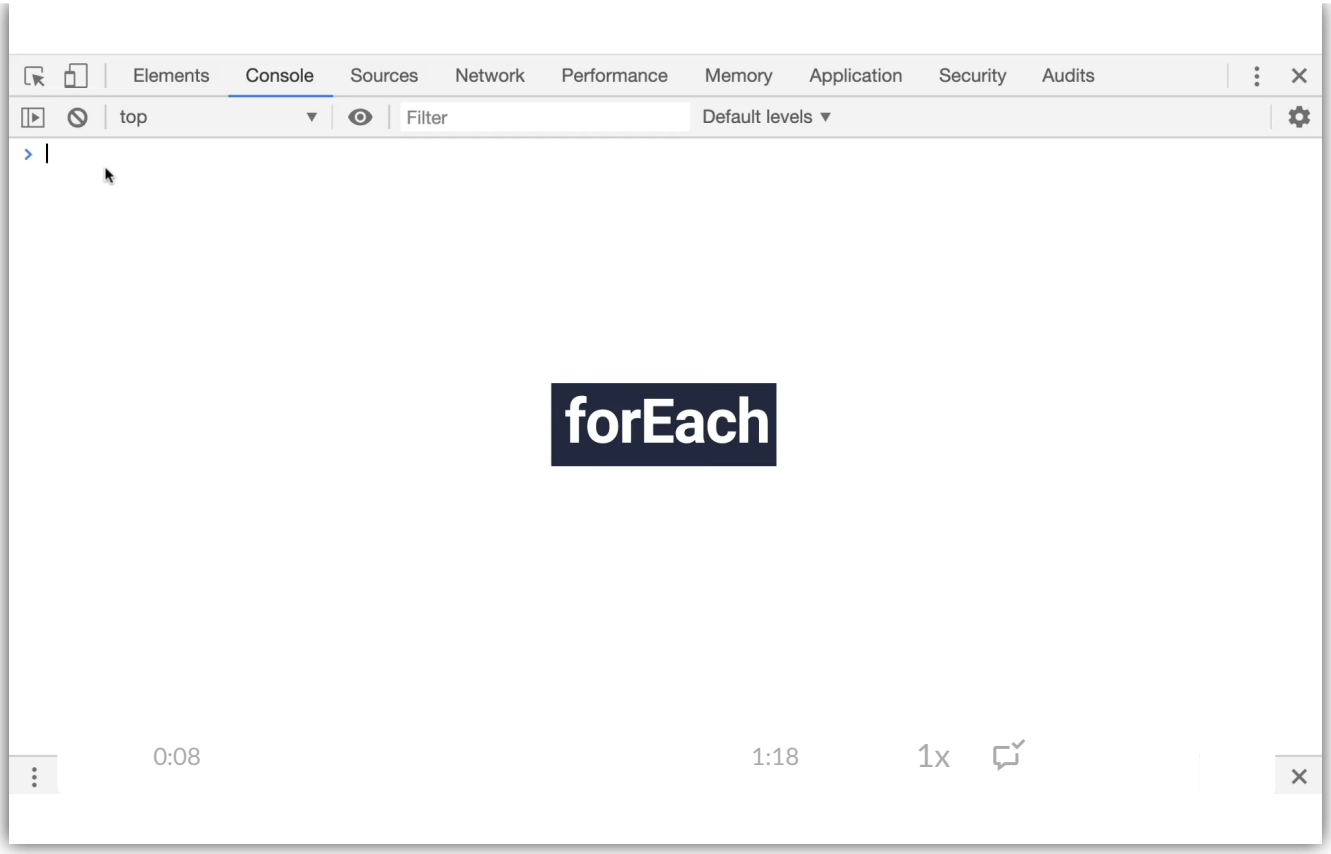
Now that we have the start of a clean table, let's apply the `forEach` function.

11.5.2: Add forEach to Your Table

Now Dana is ready to start adding data to her table. To do so, she'll need to create another function specifically for building the table. Data from the `data.js` file will be inserted into the table, row by row. This sounds like iterating through an array using a `for loopforEach`, doesn't it?

This time, we'll use a `forEach` function, which loops through the array in the same way as a `for` loop. The difference is that `forEach` works *only* with arrays. Another benefit is that `forEach` can be combined with an arrow function, once again making the code more concise and easy to read.

In the next step, we'll incorporate a `forEach` function that loops through our data array, and then adds rows of data to the table. The following video provides an overview of how `forEach` functions work.



Add the forEach Function

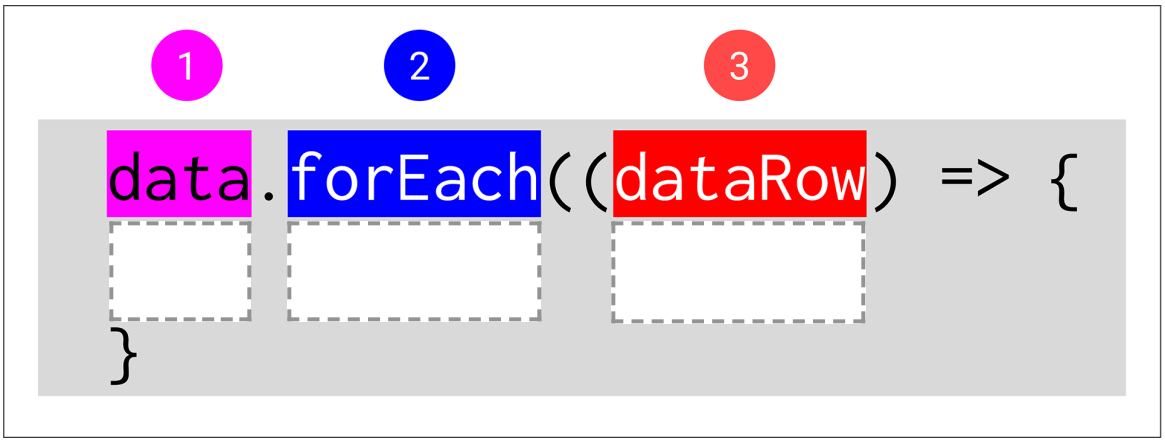
This function works in the same way as a `for` loop. In your code editor, type the following:

```
data.forEach((dataRow) => {

});
```

Notice the fat arrow? We're using an arrow function here because it's a cleaner way to write a `forEach` loop. There's nothing wrong with using a traditional `for` loop—the code would behave in the exact same manner—but it is neater and easier to read.

Match the components of the code to their actions.



- An object that references the data being imported
- A parameter that will be used as a value when the function is called
- The keywords to create a `for` loop in JavaScript

Check Answer

Finish ▸

With this new function, we have essentially chained a `for` loop to our data. We also added an argument (`dataRow`) that will represent each row of the data as we iterate through the array. Now we want to create a variable that will append a row to the table body. Within this forEach function, add the following code:

```
    let row = tbody.append("tr");
```

Note

> Notice that we're using `let` instead of `var` to declare the `row` variable. That's because this variable is limited to just this block of code. It's more appropriate to use `var` when we want the variable to be available globally, or throughout all of the code.

This single line of code is doing a lot. It tells JavaScript to find the `<tbody>` tag within the HTML and add a table row (`"tr"`).

We'll get back to HTML when it's time to display our table, but for now keep in mind that the `<tr>` tags are used for each row in a table. Each object, or UFO sighting, in the array will be wrapped in a `<tr>` tag.

Loop Through Data Rows

Next, we'll add code to loop through each field in the `dataRow` argument. These fields will become table data and will be wrapped in `<td>` tags when they're appended to the HTML table. It gets a little confusing here, but we're going to set up another function within our original function for the `forEach` loop.

Below the line where we appended table rows, we'll set up another function:

```
    Object.values(dataRow).forEach((val) => {
});
```

We're already working with an array of objects, where each object is a UFO sighting. By starting our line of code with `Object.values`, we're telling JavaScript to reference one object from the array of UFO sightings. By adding `(dataRow)` as the argument, we are saying that we want the values to go into the dataRow. We've added `forEach((val)` to specify that we want one object per row.

Let's think of it this way: we're telling our code put each sighting onto its own row of data. The `val` argument represents each item in the object, such as the location, shape, or duration.

In the next two lines of code, we'll append each value of the object to a cell in the table. In our editor, the next few lines of code will go inside our new function. Let's first create a variable to append data to a table:

```
    let cell = row.append("td");
```

With this line, we've set up the action of appending data into a table data tag (`<td>`). Now, in the next line we'll add the values.

```
    cell.text(val);
```

What does the line `cell.text(val);` accomplish?

- This is the variable that holds only each value from the object.
- It adds the key and value pair from the object.
- It adds only the key to each cell of the table.

Check Answer

Finish ▸

Let's take a look at our fully assembled function:

```
data.forEach((dataRow) => {
  let row = tbody.append("tr");
  Object.values(dataRow).forEach((val) => {
    let cell = row.append("td");
    cell.text(val);
    }
  );
});
```

With this function, we have done the following:

- Looped through each object in the array
- Appended a row to the HTML table
- Added each value from the object into a cell

For example, this is the very first object in our array:

```
{
  datetime: "1/1/2010",
  city: "benton",
  state: "ar",
  country: "us",
  shape: "circle",
  durationMinutes: "5 mins.",
  comments: "4 bright green circles high in the sky going in circles th
},
```

The code we just wrote will turn this object into a clean table like the one below.

Date	City	State	Country	Shape	Duration	Comments
1/1/2010	benton	ar	us	circle	5 mins	4 bright green circles high in the sky going in circles then one bright green light at

my front door.

Also, because we've added `forEach`, every object in the array will be added to its own row in the table. We'll be able to test it fully soon, but first we need to add filters to the table. (We'll get to this step soon.)

The complete `buildTable` function we created should match the one below. It's also a good idea to add comments to help us keep track of what our code is doing.

```
function buildTable(data) {
  // First, clear out any existing data
  tbody.html("");
  // Next, loop through each object in the data
  // and append a row and cells for each value in the row
  data.forEach((dataRow) => {
    // Append a row to the table body
    let row = tbody.append("tr");
    // Loop through each field in the dataRow and add
    // each value as a table cell (td)
    Object.values(dataRow).forEach((val) => {
      let cell = row.append("td");
      cell.text(val);
      }
    );
  });
}
```

The first step of our code is complete: we've created a table!

Add, Commit, Push

Be sure to save your work and add, commit, and push it to your repo!

11.5.3: Add Filters

The code we helped create will add every object in ou `data.js` file to the table. Bundled into one tidy package, every sighting will be available for Dana (and her readers) to view! There are a lot of objects, though, which will make the table huge! There will be hundreds of rows of sightings in the table, which is entirely too much for one person to reasonably look through and study. Therefore, the next step is to add the ability to filter the data. We'll be using D3.js to help Dana with this part.

Data-Driven Documents (D3 for short) is a JavaScript library that adds interactive functionality, such as when users click a button to filter a table. It works by "listening" for events, such as a button click, then reacts according to the code we've created.

Dana thinks that she would like to filter by date, so she'll add code to create a date filter.

We'll need to add a second function to our code that will focus on filtering the table we just built. We'll use a popular library, D3.js, to equip our website to "listen" for events, such as a user clicking a button.

In our code, we're going to use D3 to handle an action from a user, such as a button click. This means that we'll add an actual button to our HTML page to filter the table. When the button is clicked, D3 will detect the click and react accordingly. Building out user-driven data visualizations is an essential part of the data visualization job; it can be tricky at first, but oh-so-satisfying when it works! Let's get started.

Return to VS Code and our `app.js` file and start a new function. We'll name this one "handleClick" because it will be handling what to do after an input is given, such as filtering the table by date.

Let's go ahead and set up the function.

How will the new function be set up?

- `function handleClick() { }`
- `function handleClick()`
- `handleClickFunction() { }`

Check Answer

Finish ▸

Since we're adding a date function, we need to create a couple of variables to hold our date data, both filtered and unfiltered.

```
function handleClick() {
let date = d3.select("#datetime").property("value");
```

So what's going on in this code? D3 looks a little different from what we're used to seeing, but that's because it's closely linked to HTML.

The `.select()` function is a very common one used in D3. It will select the very first element that matches our selector string: "#datetime". The selector string is

the item we're telling D3.js to look for.

With `d3.select("#datetime")`, for example, we're telling D3 to look for the #datetime id in the HTML tags. We haven't created our HTML yet, but we know that the date value will be nested within tags that have an id of "datetime."

By chaining `.property("value");` to the `d3.select` function, we're telling D3 not only to look for where our date values are stored on the webpage, but to actually grab that information and hold it in the "date" variable.

Now we need to set a default filter and save it to a new variable. Our default filter will actually be the original table data because we want users to refine their search on their own terms. Let's add the new variable on the next line.

```
let filteredData = tableData;
```

Here's a variable we haven't seen in a while: tableData. This is the original data as imported from our `data.js` file. By setting the filteredData variable to our raw data, we're basically using it as a blank slate. The function we're working on right now will be run each time the filter button is clicked on the website. If no date has been entered as a filter, then all of the data will be returned instead.

The next step is to check for a date filter using an `if` statement.

11.5.4: Use the “If” Statement

Navigating through functions and `for` loops in JavaScript has helped Dana feel more comfortable with coding in this new language. The next part she’ll start working on is adding an `if` statement.

The `if` statement in JavaScript is similar to the Pythonic `if` statement, though the syntax is a little alien in comparison. Dana will need to explore how to create `if` statements in JavaScript and then incorporate it into her code as part of the filtering component.

Much like in Python, an `if` statement in JavaScript will check for conditions before executing the code. Our code will check for a date filter, so our `if` statement should read as follows; “If there is a date already set, then use that date as a filter. If not, then return the default data.”

Look at a basic JavaScript `if` statement, but just for practice as this won’t be part of our `app.js` code.

```
// if-statement syntax
if ( condition ) { code to execute }
```

In its most basic form, the `if` statement looks similar to a function. Write pseudocode about what we want our code to do.

```
// pseudocode practice
if (a date is entered) {
Filter the default data to show only the date entered
};
```

That makes a little more sense. We want JavaScript to check for a date. If one is present, we want it to return only the data with that date. Now return to our app.js file to add our `if` statement.

```
    if (date) {
filteredData = filteredData.filter(row => row.datetime === date);
    };
```

Take a closer look at the line that's inside our `if-statement`.

We just created the `filteredData` variable. What are we doing with it in this line: `filteredData = filteredData.filter(row => row.datetime === date);` ?

- We're assigning a new date based on the original data.
- The code is searching for user input to filter by.
- We're applying a filter method that will match the datetime value to the filtered date value.

Check Answer

Finish ▸

Take a look at the syntax for the `.filter()` method: `row => row.datetime === date);`. This line is what applies the filter to the table data. It's basically saying, "Show only the rows where the date is equal to the date filter we created above." The triple equal signs test for equality, meaning that the date in the table has to match our filter exactly.

Important

There are two ways to test for equality in JavaScript: `==` and `===`. While they look similar, there are differences. A triple equal sign (`===`) is checking for **strict equality**. This means that the type and value have to match perfectly.

A double equals sign (`==`) is checking for **loose equality**. This means that the type and value are loosely matched. For more information about equality in JavaScript, read **JavaScript — Double Equals vs. Triple Equals (https://codeburst.io/javascript-double-equals-vs-triple-equals-61d4ce5a121a)** .

When we look at our complete `if` statement, it should appear as follows:

```
if (date) {
  filteredData = filteredData.filter(row => row.datetime === date);
};
```

This is great! Our `handleClick()` function tells the code what to do when an event occurs (such as someone clicking a filter button), and it can apply that filtered data using an `if` statement. Being able to do all of this is great, especially since it involves creating functions written in a syntax that isn't the easiest to learn. There is one more step to complete with this function, though: building the table using the filtered data.

Build the Filtered Table

Thankfully, we've already set up a function to build a table: `buildTable();`. Now we just need to call it. Remember, we're building the function with the filtered data, so we'll use that variable as our argument.

Under our `if-statement`, let's call the `buildTable` function.

How will we call the function to build our table with the filter in place?

- `function buildTable(data);`
- `buildTable(filteredData);`
- `function buildTable(filteredData);`
- `buildTable(data);`

Check Answer

Finish ▸

After we pass `filteredData` in as our new argument, our full `handleClick()` function should look like the one below:

```
function handleClick() {
  // Grab the datetime value from the filter
  let date = d3.select("#datetime").property("value");
  let filteredData = tableData;
  // Check to see if a date was entered and filter the
  // data using that date.
  if (date) {
    // Apply `filter` to the table data to only keep the
    // rows where the `datetime` value matches the filter value
    filteredData = filteredData.filter(row => row.datetime === date);
  };
  // Rebuild the table using the filtered data
  // @NOTE: If no date was entered, then filteredData will
  // just be the original tableData.
  buildTable(filteredData);
};
```

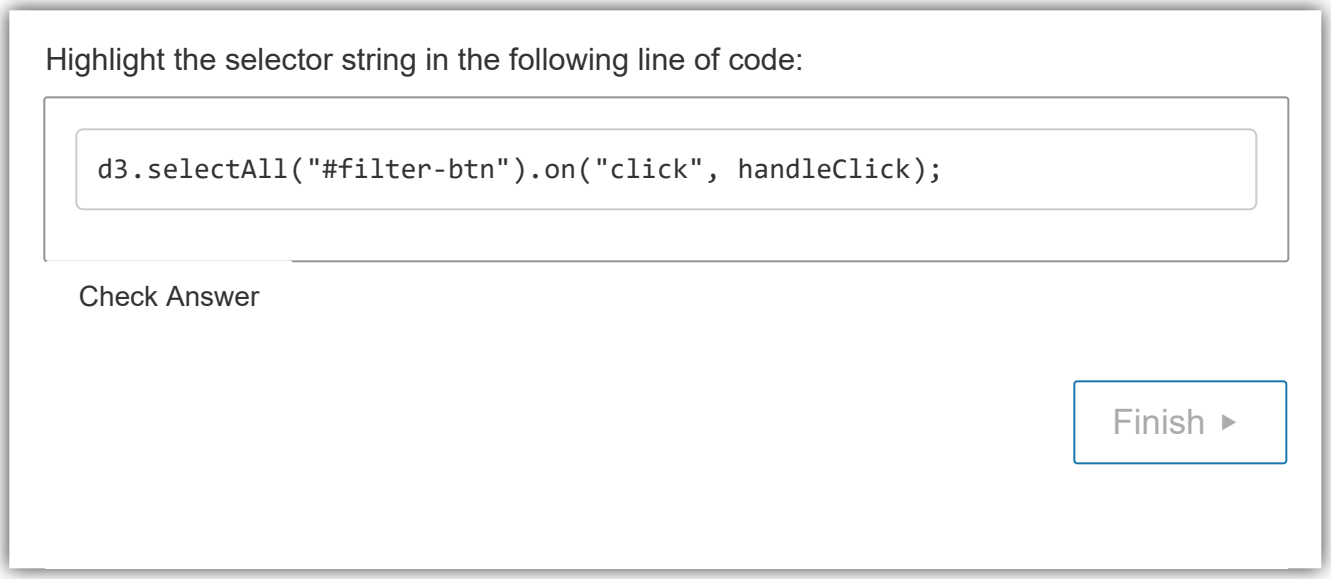
Listen for the Event

Our code is almost ready to be attached to the HTML component of our webpage. There are just a couple of loose ends to tie up. One is the clicking that will happen when someone filters the table. We have a function that *handles* a click, but how does the code know when a click happens?

Another aspect of D3.js is that it can listen for events that occur on a webpage, such as a button click. The next code we add will be tied to the filter button we'll build on our webpage.

Under our `handleClick()` function, add the following line of code:

```
d3.select("#filter-btn").on("click", handleClick);
```



Our selector string contains the id for another HTML tag. (We'll assign a unique id to a button element in the HTML called "filter-btn".) This time it'll be included in the button tags we create for our filter button. By adding this, we're linking our code directly to the filter button. Also, by adding `.on("click", handleClick);`, we're telling D3 to execute our `handleClick()` function when the button with an id of `filter-btn` is clicked.

> **Note**
> A "click" isn't the only event that D3.js can listen for—there are a variety of actions that can be listened for and handled. For example, a tooltip displays when you place your mouse over a specific element on a webpage.

> Events aren't limited to mouse events, either. They can include keyboard, text composition, forms—the list is lengthy and some of the events are quite advanced.

Build the Final Table

There is only a single step left before we can build the HTML component of the webpage: making sure the table loads as soon as the page does. Dana's readers will need to see the original table to even begin to use the filter we've set up. At the very end of the code, we'll call our `buildTable` function once more—this time using the original data we've imported. Type the following code:

```
buildTable(tableData);
```

Once this function is called, it will create a basic table filled with row upon row of unfiltered data pulled straight from our array.

All together, our code in the `app.js` file will look as follows:

```
function handleClick() {
  // Grab the datetime value from the filter
  let date = d3.select("#datetime").property("value");
  let filteredData = tableData;
  // Check to see if a date was entered and filter the
  // data using that date.
  if (date) {
    // Apply `filter` to the table data to only keep the
    // rows where the `datetime` value matches the filter value
    filteredData = filteredData.filter(row => row.datetime === date);
  }
  // Rebuild the table using the filtered data
  // @NOTE: If no date was entered, then filteredData will
  // just be the original tableData.
```

```
  buildTable(filteredData);
}
// Attach an event to listen for the form button
d3.selectAll("#filter-btn").on("click", handleClick);
// Build the table when the page loads
buildTable(tableData);
```

Now you're ready to start building your webpage!

> ### Add, Commit, Push
>
> Make sure to save your work and add, commit, and push it to your repo!

11.6.1: Bootstrap Components

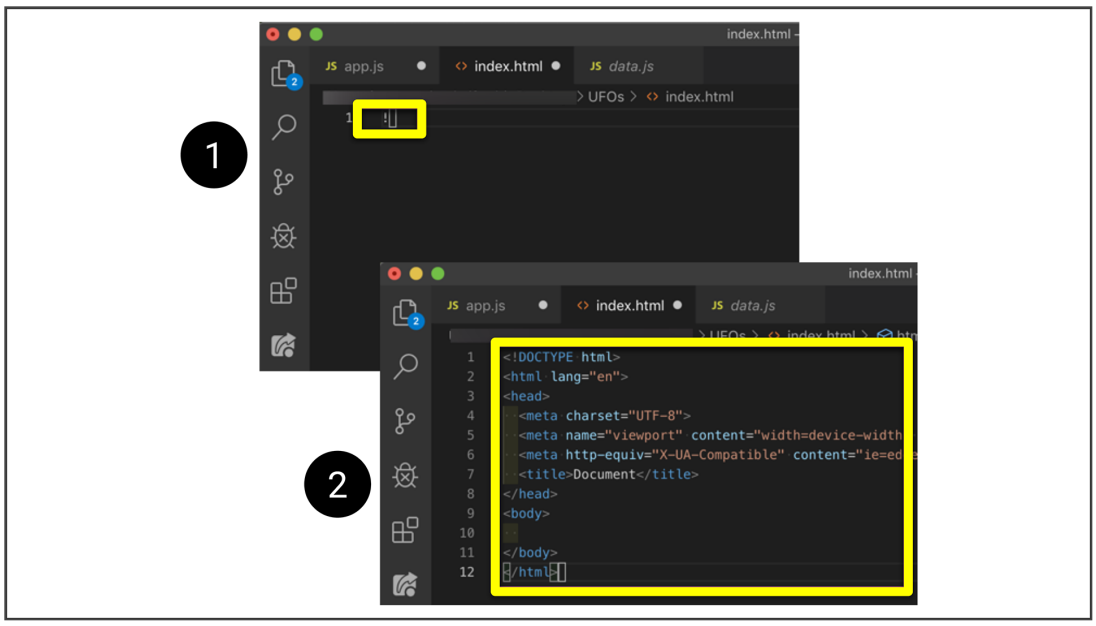
With our help, Dana has assembled code that will not only build a functional table but will add the ability to filter it as well. That's definitely an accomplishment—JavaScript isn't an easy language to get the hang of. But even though she thinks her code is structurally sound and will work without a hitch, she still needs to test it. To do so, it's time to build the HTML page Dana will be showcasing her work on.

She'll need to keep the different ids she created in app.js in mind while she pieces this webpage together. There will be a button (`#filter-btn`) somewhere as well as a `#datetime` id nestled into some tags. She'll need to build the base of the table so that the code we helped construct knows where to put the table's data. This means that the columns and rows will all need to be defined manually.

Thankfully, Dana still has the storyboard she assembled earlier, which will speed up the assembling process.

The time has come to build the webpage. This is known territory for Dana, so she's excited to take a short brain break and work with the more familiar tools of Bootstrap and HTML. It's also a test to see if the JavaScript code is working.

First, we'll need to go back to the `index.html` file that we created earlier. If you haven't already, open that file in VS Code. Next, use a shortcut to autofill the basic HTML layout by typing an exclamation mark on the first line, then press enter on your keyboard.



With the basics already completed for us, we can start customizing. First, we can change the title of the document to "UFO Finder."

Continue the setup by adding a link to Bootstrap's content delivery network (CDN). Under the title of the document, add this code to the link tag:

```
<link
      rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap
      integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAw
      crossorigin="anonymous"
    />
```

Next, add a link to our stylesheet under the link to Bootstrap's CDN. Since we created a stylesheet and index.html file at the same time, we'll just link to the style.css file that's in our css folder.

```
<link rel="stylesheet" href="static/css/style.css">
```

Now we can begin setting up the body of the HTML, where we'll store our components. To get started, within the `<body />` tags, add a `<div />` with a class of `"wrapper."`

```
<body>
    <div class="wrapper">

    </div>
</body>
```

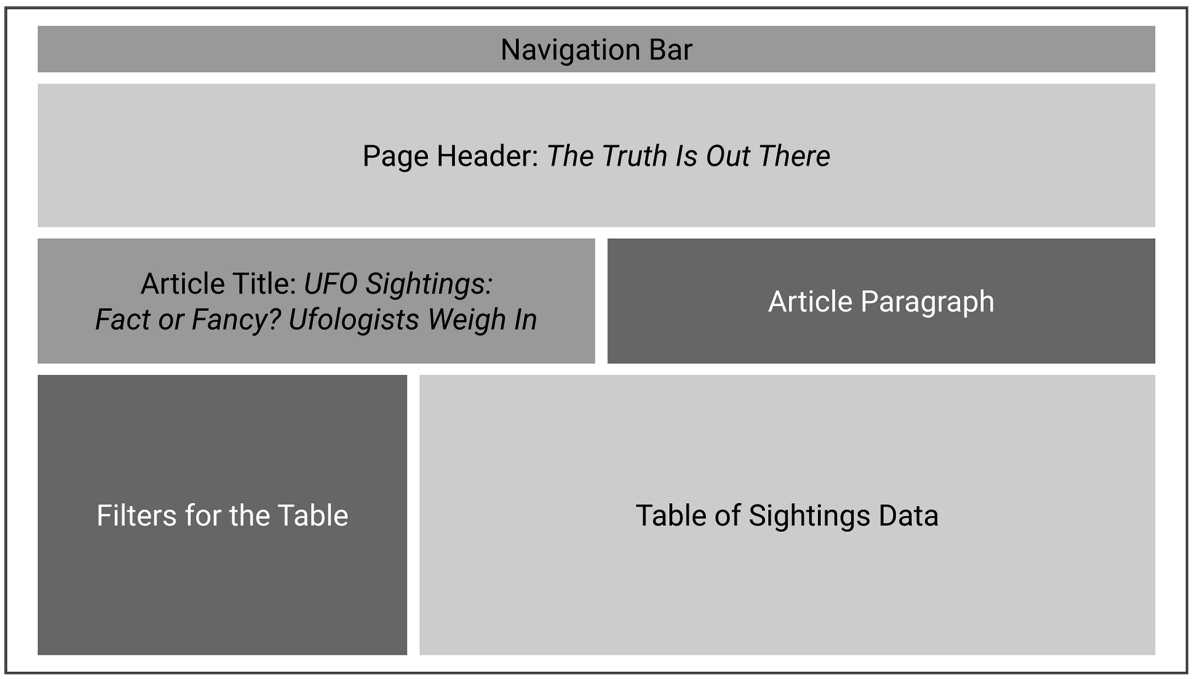
The wrapper class adds a bit of extra functionality to Bootstrap. It helps group the elements (e.g., title, paragraph, table, and filters) and specifies the styling in our stylesheet.

Now we can start adding the individual components.

The Bootstrap Grid System

Take a look at the storyboard we helped Dana create and consider how to employ the Bootstrap grid system.

> **Rewind**
> The Bootstrap grid system helps organize a webpage's content into containers, rows, and columns. A page can contain up to 12 columns per row and as many rows and containers as needed for content display. It also allows for a responsive webpage that will resize for the viewing area or screen size.



There are many different components to build. Let's start with the first one we see: the navigation bar (or navbar), where we'll add the functionality to reset our table filters.

Build the Navbar

Within the wrapper we created earlier, we'll add a new component: a nav tag with a class of `"navbar navbar-expand-lg."`

```
    <nav class="navbar navbar-expand-lg">

    </nav>
```

These classes are specific to Bootstrap's built-in styling; `"navbar"` indicates to Bootstrap that we want a component that fits across the top of the page. Specifying `"navbar-expand-lg"` provides extra responsiveness to the default navbar behavior. When the viewing area is reduced from a large to a smaller screen, the navbar will collapse or resize itself smoothly.

Now we need to add functionality to our navbar. In this case, we don't need to redirect readers to another section of the webpage. Instead, we want to reset the webpage after a filter has been applied to the table. This is accomplished by linking to the homepage, `index.html`.

To add a link, we'll nest an `<a />` tag within the `<nav />` tags.

```
<nav class="navbar navbar-expand-lg">
  <a class="navbar-brand" href="index.html">UFO Sightings</a>
</nav>
```

There are a few things happening within this new tag.

Why was `"navbar-brand"` added to the `<a />` tag?

- To specify that it's still inside the `nav` tag, each additional element added will need to be flagged in a similar manner.
- Because Dana is adding branding to the navbar.
- Because `"navbar-brand"` is a type of default styling that helps with the site's aesthetics.

Check Answer

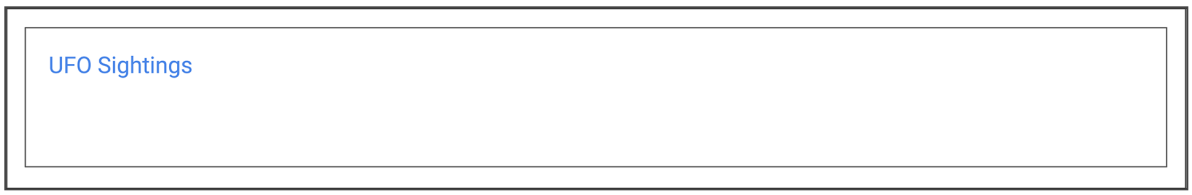
Finish ▸

After adding `"navbar-brand"` to the tag (so there is less styling to worry about later), we also added an `href` that points to the `index.html` file we're working on. When a user clicks on that link, the page will reload and the default unfiltered table will appear, ready for new input.

We also need to display text in the navbar and complete the link. Dana has added "UFO Sightings" as the text for now.

Let's test to make sure our link to Bootstrap and the navbar is implemented correctly. Find your `index.html` file in your repo and open it with your default

browser. The website should have a line of text that reads “UFO Sightings” in the top left corner.



The design is plain now, but we’ll return later to customize.

Add the Jumbotron

The first element, a navbar, has been added. Next, add the jumbotron. Because it’s a new element and completely separate from the navbar, we want to add the new code beneath the existing code.

Let’s set up the jumbotron by adding `div` tags, then the proper class.

Which of the following options sets up the jumbotron correctly?

- `<div>Jumbotron</div>`
- `<div jumbotron></div>`
- `<div class=”jumbotron”><div>`
- `<jumbotron></jumbotron>`

Check Answer

Finish ▸

Bootstrap looks for certain classes within HTML tags to indicate where to apply styling, such as by adding a `"jumbotron"` class to a `div` tag. Text nested within these tags will have visual enhancements automatically added. For example, a header tag nested within a jumbotron will be larger and bolder than a header tag

on its own. In our code editor, let's add a jumbotron with a header that reads "The Truth Is Out There."

```
<div class="jumbotron">
  <h1>The Truth Is Out There</h1>
</div>
```

After saving this file, refresh the page we have open in our browser.

It's still rather plain, but we'll spruce it up after we finish assembling the other components.

Add the Article Title and Paragraph

So far, the webpage has the navigation established and a header that pops (and even more so after we customize it). Now we'll start getting into the content that Dana wants to display. In this section, we'll add the article title and paragraph. According to our storyboard, we'll need to utilize the grid system, which will let us assign screen space to each element.

The grid system, consisting of containers, rows, and columns, will need to be set up in a certain order: first the container, then a row, followed by how many columns each element will require. Create the container and row first.

In your text editor, create a `div` with a class of `"container-fluid,"` then nest a row within it.

In Bootstrap, what does the `"container-fluid"` class do? Select all that apply:

- [ ] It creates a container for other elements.
- [ ] It adds Bootstrap styling to the element(s) inside it, such as adding a background color and specific font.
- [ ] Element(s) inside the container will resize as needed to fully fit the width of the browser.
- [ ] The `div` with a class of `"container-fluid"` is given a fixed-width.

Check Answer

Finish ▸

Adding `"container-fluid"` to the `div` will ensure that both elements we're adding will span the width of the viewport. The `"row"` class makes sure that the title and paragraph will align neatly along the page.

```
<div class="container-fluid">
    <div class="row">
    </div>
</div>
```

Now we can add our columns. According to our storyboard, the title requires less width than the paragraph.

Let's assign four columns to the article title and the remaining eight to the paragraph. Remember that each element will be within its own `div`.

```
        <div class="col-md-4">
```

```
        </div>
        <div class="col-md-8">

        </div>
```

Now that the scaffolding is in place to hold the title and article, we can insert the title and paragraph Dana has chosen. Using an `<h3 />` tag, nest the article title ("UFO Sightings—Are They for Real? Ufologists Weigh In") in the first column.

```
<h3>UFO Sightings: Fact or Fancy? <small>Ufologists Weigh In</small></h3>
```

> **Note**
>
> The `<small>` tag we've nested in will add a little bit of extra styling out of the box, too. Adding it will help de-emphasize the second portion of the title.

In the second `div` we added, the one that uses eight columns, let's add Dana's article paragraph. Here is what needs to be added:

Are we alone in the universe? For millennia, humans have turned to the sky to answer this question. Now, thanks to research generously funded by W. Avy, a UFO-enthusiast and amateur ufologist, we can supplement our sky-searching with data analysis.

"The release of this analysis is well-timed: It coincides with the celebration of World UFO Day, which is a moment for ufologists around the world to connect, relax, and sample a range of UFO-themed snacks," said Dr. Ursula F. Olivier, the world's preeminent expert on circular sightings. "Citizen-scientists can be especially helpful in both cataloguing sightings—which is hugely helpful for us in our search for aliens—and in helping us celebrate the work that has already been done, such as this data visualization project, which will help us raise awareness of the ubiquity of sightings!"

Not everyone is ready to celebrate, however. Local CEO and vocal anti-alien activist V. Isualize reached out to our reporters to go on record as firmly opposed

to any attempts to provide access to this data. “If there are aliens, they certainly would like to be left alone,” she stated, before directing us to the Leave Aliens Alone (LAA) community engagement initiative she founded and funds.

So what do YOU think? Are we alone in the universe? Are aliens trying to contact us, or do they want to be left alone? Dig through the data yourself, and let us know what you see.

Our final code for this section should be similar to what’s displayed below.

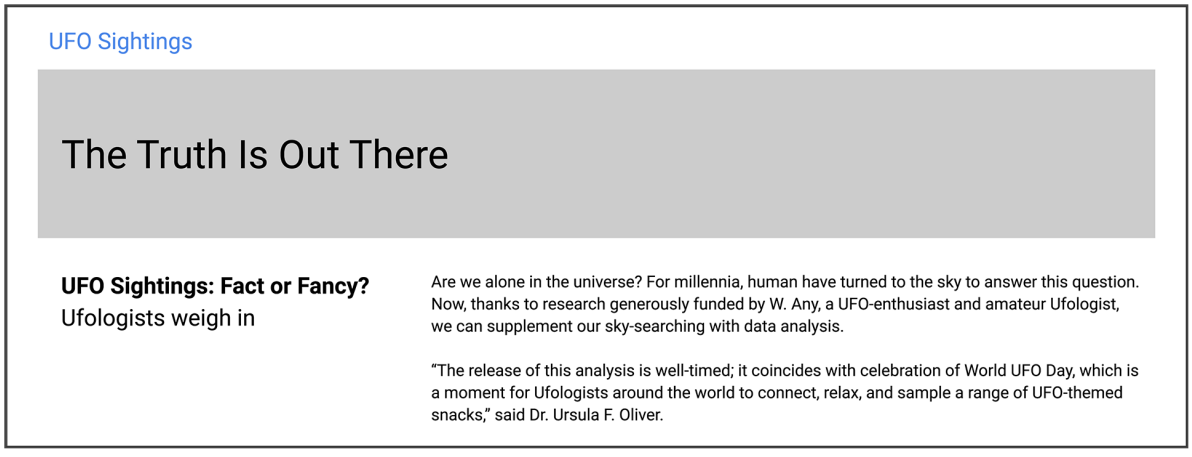
```
    <div class="container-fluid">
      <div class="row">
        <div class="col-md-4">
          <h3>UFO Sightings: Fact of Fancy? <small>Ufologists Weigh In<
        </div>
        <div class="col-md-8">
          <p>
            Are we alone in the universe? For millennia, humans have tu



          </p>
        </div>

      </div>
    </div>
```

When we save this code and refresh our webpage, we’ll see how the webpage is really starting to come together.



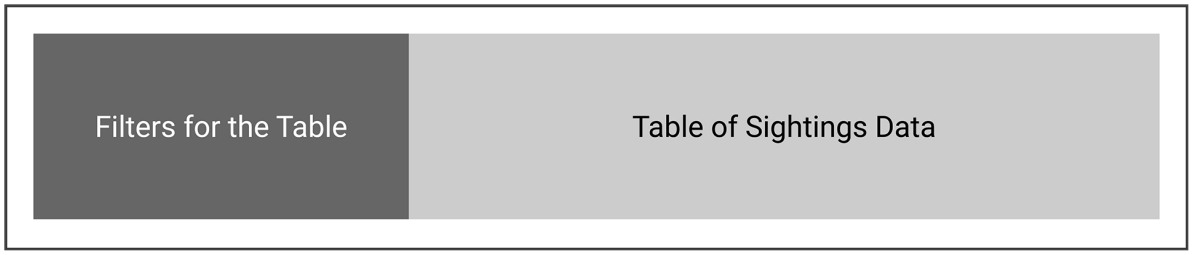
Create the Table Filter

The next section of the webpage will tie together the JavaScript we've been helping Dana put together: We're going to build the section for the filter. The ids we created in our JavaScript code (`#filter-btn` and `#datetime`) will come into play here.

First, we need to set up the next row that will hold the filter section and the table.

> **Skill Drill**
> Create a new fluid container, and then nest a row inside it. This row will be where we store our filter and the data table.

When the new row has been created, we need to determine how many columns to designate for the filter section and how many for the table. Let's look at our storyboard again.



Compared to the previous row, the filters section looks like it will require less column space than the article title. Let's create a `div` with space for three columns and a nine-column `div` for the table.s

```
      <div class="container-fluid">
        <div class="row">
          <div class="col-md-3">

          </div>
          <div class="col-md-9">

          </div>
```

```
        </div>
      </div>
```

Let's build the filter first. This is where we'll be accepting user input, so we need to use the HTML-specific field: a form.

First, we'll add a `form` tag, then build the form by nesting additional elements within it. Let's give the form a name so that users will know what it's for. After the `<form />` tag, add a `<p />` tag and the text `"Filter Search"` to that.

```
            <form>
              <p>Filter Search</p>
            </form>
```

With this, we know what this new component is, but now we need to add what Dana's readers will be searching for. Our JavaScript code has a filter setup to search by date, so we'll need to add an input box for a date. We'll also need the button that we referenced (`#filter-btn`) so that searches can be executed.

To add these items as cleanly as possible, we're going to create a set of list tags, nested within an unordered list tag. We'll also include some Bootstrap classes to keep things extra neat.

In your code editor, let's begin by starting the unordered list. We're going to give this a class of `"list-group."` Using this specific class lets Bootstrap apply predetermined styling to the list. We can spruce it up further if we want to after we're done.

```
<ul class="list-group">

</ul>
```

Next we need to add the list items: one for the input, one for the button. Each `<li />` tag will have a class of `"list-group-item."`

```
<li class="list-group-item">

</li>
<li class="list-group-item">

</li>
```

Now let's add the date input field in the first `<li />` tag. We'll add two new HTML components here: label and input. The label will be used as a prompt to encourage users to input a date. The input field is where users will complete the input.

In your code editor, add the label tag with the following modifications:

```
<label for="date">Enter Date</label>
```

This label represents a caption for the date item. The text `Enter Date` serves as the actual label.

On the next line, let's add the input.

```
<input type="text" placeholder="1/10/2010" id="datetime" />
```

There are three things to keep in mind when looking at this input:

1. The `type="text"` means that the code will look for text to be input.
2. The placeholder is an example of a date to search, so users know both the location and the format to use when inputting a date.
3. The `id="datetime"` is what our JavaScript code will look for when the button is clicked and the function is executed.

Now we need to add our button. In the next `<li />` we'll add a button tag with a few additional attributes: the id we defined in our JavaScript code (`#filter-btn`), a type, and a class. Let's add this as well.

```
<button id="filter-btn" type="button" class="btn btn-default">Filter Tabl
```

When the button is clicked, the input from earlier is picked up by our JavaScript code and then applied to the filter.

`type="button"` tells the browser that, by default, the button does nothing. However, our custom JavaScript script will overwrite the default behavior—as if the button is waiting for instruction.

The `class="btn btn-default"` attribute adds some Bootstrap styling to the mix to help keep the element neat and tidy. Finally, we've named our button "`Filter Table,`" but nesting it between the opening and closing button tags.

Prep the Table for Data

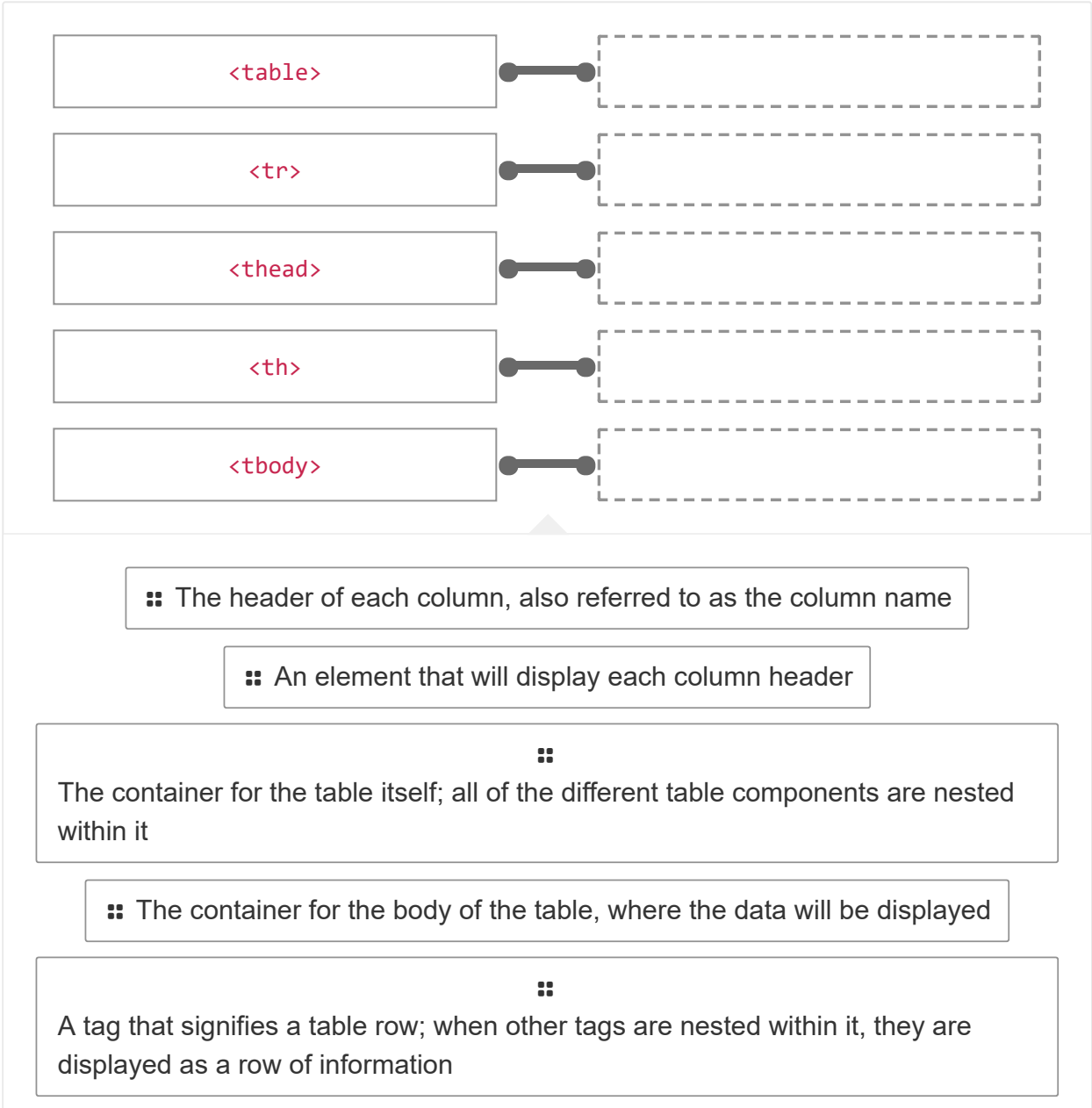
With the filter in place, we're now able to build the table. In our app.js script, we use D3 to select the "`tbody`" HTML tag. We're going to build that component and link the JavaScript and HTML files.

The table and filter components are both inside the same container, so we only need to construct the table HTML.

An HTML table has several nested layers:

1. `<table>`
    - `<thead>`
        - `<tr>`
            - `<th>`
    - `<tbody>`

Match the table tags to their description.



Check Answer

Each tag present in an HTML table is used to either designate a section of the table, such as the `<thead>` tag, or it holds information that will be displayed. If one of the tags is missing or out of order, the entire table may not function correctly (or be assembled at all). Let's start constructing our table and adding information to it.

In our code editor, let's begin by typing out the nested tags.

```
        <div class="col-md-9">
          <table class="table table-striped">
            <thead>
              <tr>
                <th></th>
              </tr>
            </thead>
            <tbody></tbody>
          </table>
        </div>
```

We've included more Bootstrap styling by adding the classes `"table table-striped"` to the table tag. This will present a table that is slightly striped to give variation between the rows of data.

The `<thead>` tag will have a few nested tags within it. This is our table setup: All of the information displayed as headers will be added into this tag and its nested components.

The `<tr>` component signifies that everything nested within it will be displayed as a row of data. We will immediately see its use as we add each column header with the `<th>` tag.

Take another quick look at one of our data.js objects.

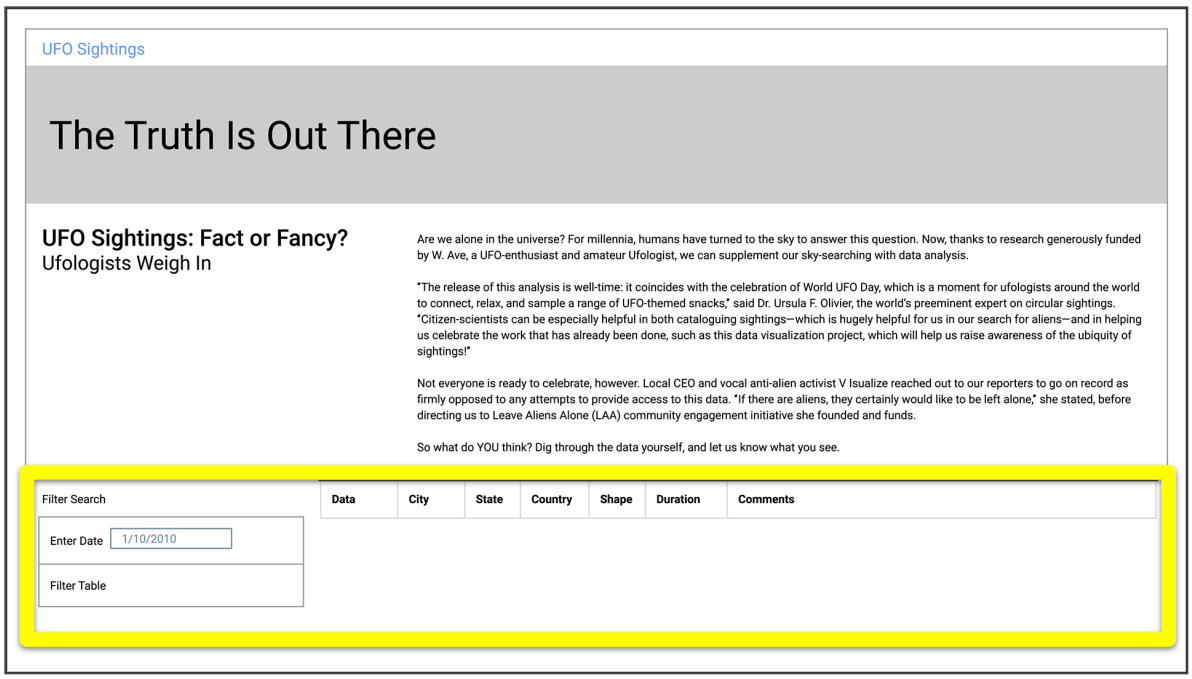
```
{
  datetime: "1/1/2010",
  city: "benton",
  state: "ar",
  country: "us",
  shape: "circle",
  durationMinutes: "5 mins.",
  comments: "4 bright green circles high in the sky going in circles th
},
```

The information in our object is present as key-value pairs (KVPs). Each object will have the same key, but different values—these keys (such as `datetime`, `city`, and `state`) will be our table headers.

Back in our HTML code, we will need to add a `<th />` tag for each table header. Let's clean up the text a bit by using proper capitalization and spacing.

```
<th>Date</th>
<th>City</th>
<th>State</th>
<th>Country</th>
<th>Shape</th>
<th>Duration</th>
<th>Comments</th>
```

Save the file, then refresh the `index.html` page you have open in your browser—the only thing missing is the actual table data from the data.js file.



Add, Commit, Push

Be sure to save your work and add, commit, and push it to your repo!

11.6.2: Add the Data

Dana's page is starting to really come together. The layout from the storyboard has transferred directly to the HTML, making the construction fairly seamless. There was still quite a bit of nesting and manual entry going on, but the overall result is a clean page, ready to display data.

The next step in getting Dana's page viewer-ready is to link `D3.js`, `app.js`, and `data.js` to the HTML.

The UFO webpage looks nice and clean and appears to be functioning well, but the only way to truly test it is to tie it together with the JavaScript code we created earlier. We'll tie them together by adding `<script />` tags then linking to the file's location. This is very similar to when we added a link to our stylesheet. Only this time, the links to our scripts will be at the bottom of the page.

> **Important**
> When adding multiple `<script />` links to a webpage, the order matters. The order we link our files will be the order they are executed. If we link app.js before data.js, then the app will try to build the table before the data has loaded. This will generate an error and break the table.

At the bottom of the page, under the last `<div />` tag, we will need to add our scripts. There are three we need to include, in the following order:

1. A link to a D3.js CDN
2. The file path to data.js
3. The file path to app.js

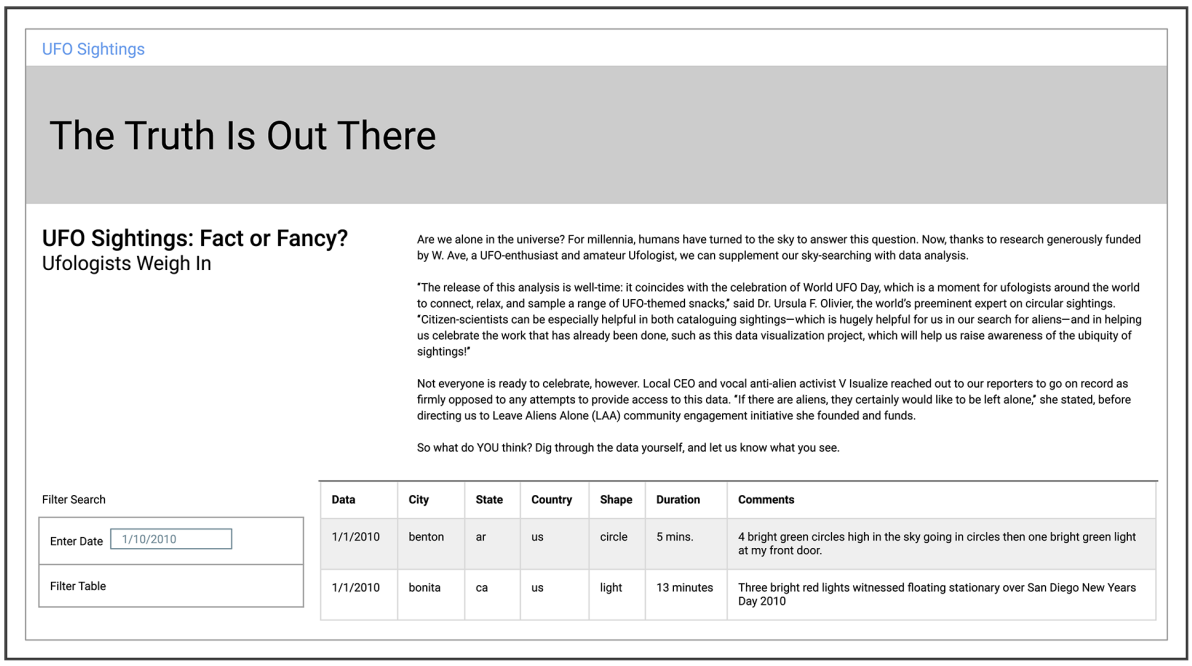
These will each be added via a script tag. Let's add them now.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.11.0/d3.js"></
<script src="static/js/data.js"></script>
<script src="static/js/app.js"></script>
```

Adding the link to D3.js allows the library to "listen" in on our code, or react to user input. For example, if we did not add this link, our `d3.select` section of code in `app.js` wouldn't know where to insert data.

We linked to `data.js` next because the UFO sightings data needs to be loaded before it can be accessed, as we do in the `app.js` script.

Once the file has been saved, return to the page you opened in your browser and refresh it. The table should now be filled to the brim with UFO sightings data. Even better, the filter button should work as intended.



When you filter the table by the date “1/13/2010,” how many results are returned? To reset the table, remember to click the “UFO Sightings” link in the navbar to reload the page.

- Two
- Three
- Five

Check Answer

Finish ▸

11.6.3: Customize the Page with CSS

Dana wants to customize her page by adding specific CSS components to the stylesheet, such as applying a background color to the page and adding an image to the jumbotron. Although it's neat and tidy in its current form, it's not very attention-grabbing.

These last few steps are all she needs to wrap up her research and publish it, so she's looking forward to adding that last bit of professional polish.

Now it's time to spruce the page up a bit. Dana wants to make use of some of the built-in styling that Bootstrap offers, but she will also need to code some of the customizations by altering the CSS associated with her webpage. The CSS customizations will take place in the style.css file we created earlier.

CSS syntax, as with all coding languages, is very exact. Each customization entry follows a pattern. First, the area of customization is set, usually identified by its element, class attribute, or id attribute. A colon follows, then the value of the customization, and a semicolon completes the line of code.

It's very similar to KVPs in an object; for example, a line that reads "`background-color: #292b2c;`" `background-color` is the key, and the number following it is the value. This particular combination will generate a dark background with light colored text. Let's use this syntax to apply a light font color to the webpage.

In our style.css sheet, enter the following lines of code:

```
body {

  color: #f7f7f7;
}
```

This block of code provides our webpage with specific styling instructions: We want a light text (`color: #f7f7f7;`) applied to the body (`body {}`) of the page.

We chose to apply the styling to the body of the page because it is one of the outermost elements—every other visual element lies within the `<body>` of the page. If we were to replace "`body`" with "`h1`," then only the `h1` elements would be affected. Similarly, because we want to apply the same styling to the entire page, we can add a "bg-dark" class to our body element. In our `index.html` file, update the opening `<body>` tag so it looks like the line below:

```
<body class="bg-dark">
```

This will apply a dark background to the entire page. We'll be adding this class to most of our elements so they all have the same styling. This applies a cleaner, more cohesive look to the webpage.

Update the Navbar

Let's also make use of Bootstrap's built-in styling, starting with the navbar. In the index.html file, let's add `navbar-dark bg-dark` to the `<navbar>` classes, like so:

```
<nav class="navbar navbar-dark bg-dark navbar-expand-lg">
```

This will apply a dark background and light lettering to the navbar for us, without any extra styling in the CSS stylesheet.

Customize the Jumbotron

Next, update the jumbotron to something more appealing by changing the font and background. This time, we'll add a Bootstrap class called `"display-4"` to the `<h1>` element inside the jumbotron, which will change the font style and size for us.

```
<div class="jumbotron">
  <h1 class="display-4">The Truth is Out There</h1>
</div>
```

Now let's make the background more interesting. Dana has chosen a space image to use as a background instead of a solid color; download it and save it to the "images" folder you created in your repo.

Download nasa.jpg
(https://courses.bootcampspot.com/courses/138/files/14458/download?wrap=1)

To assign this as our new background, we'll need to use our style.css sheet. Open that file in your code editor. What we'll do next is create an entry that will style only the jumbotron element.

Let's take a look at the CSS code that will style the jumbotron.

```
.jumbotron {
  background-image: url("../images/nasa.jpg");
  background-size: 100% 100%;
  text-align: center;
}
```

This code accomplishes a few things:

1. `.jumbotron` tells the stylesheet to apply the changes only to divs with a class of `"jumbotron."`
2. We specify a background image, then tell the code where the image is stored.

3. The size of the background is set to span the entire width and height of the element, no matter the size of the viewport.
4. We've aligned the text to be in the center of the element.

Customize the Form

The form element in the HTML will also need a small update: The text is now white, but the form's background is also white by default. Identifying what the form is telling us is difficult in its current state, so let's update the font color.

First, let's look at the form element in our HTML to identify which one we'll apply the styling to.

```
<form>
 <p>Filter Search</p>
 <ul>
   <li>
     <label for="date">Enter Date</label>
     <input type="text" placeholder="1/10/2010" id="datetime" />
   </li>
   <li>
      <button id="filter-btn" type="button" class="btn btn-default">
        Filter Table
      </button>
   </li>
 </ul>
</form>
```

We want this element to match the dark theme we've applied to the rest of the page, so we'll add the class "bg-dark" to our form element.

Let's also style the list element and the button to match the rest of the theme. We'll add a couple of different classes here to take advantage of Bootstrap's built-in styling.

Skill Drill

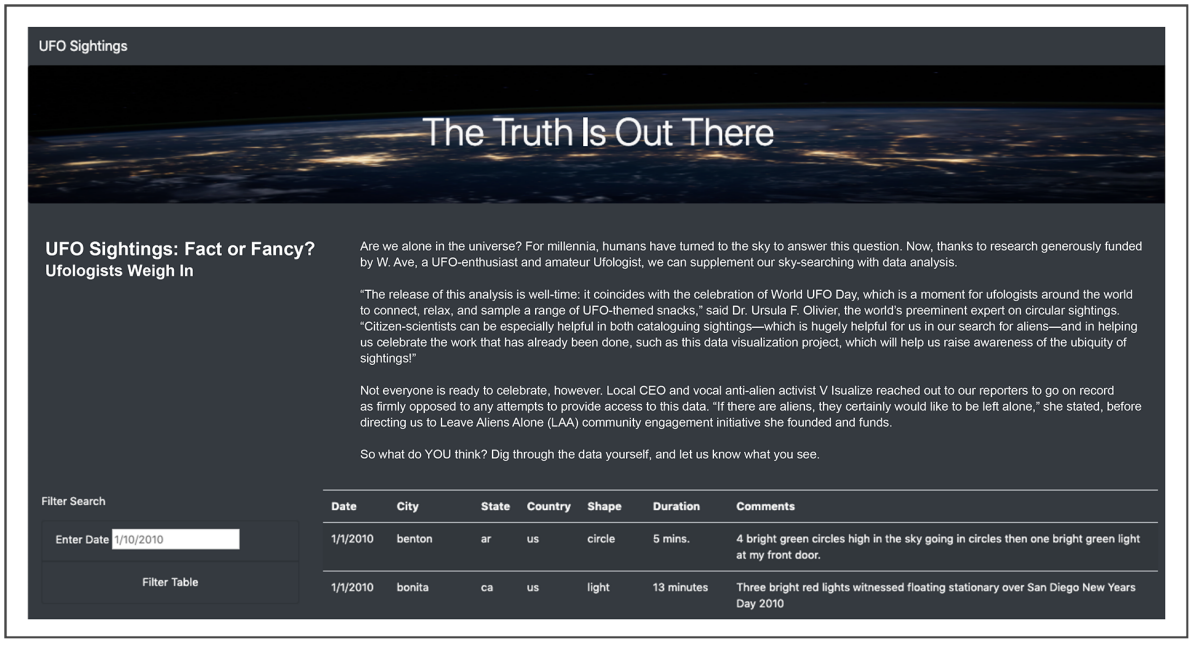
Add the following classes to the indicated elements:

1. `"bg-dark"` to the opening `<form>`, `<ul>`, and each `<li>` tag.
2. `"list-group"` to the opening `<ul>` tag.
3. `"list-group-item"` to each `<li>` opening tag.

Update the button element to use a class of `"btn-dark"` instead of `"btn-default"`.

By adding these specific classes, we have incorporated the same style and colors we've already selected (bg-dark) and also cleaned up our list elements. Instead of a bullet appearing for each list item, we now have evenly spaced rows. Additionally, the same font has been applied to the entire form. Changing the button class to "btn-dark" also matches the button's appearance to the rest of the page.

The final result is a dark page with white font -- a clean, visually appealing page where each element meshes well with the next.



Module 11 Challenge

Submit Assignment

Due Mar 29 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> The website has been finished, and the article is ready to go, but after a final look at her work, Dana feels that it could be improved even more. Instead of filtering by a single item, why not provide her readers with the ability to filter by multiple factors? The ability to search a city or country or even combine a date with a location would make Dana's application that much more robust.

In this challenge, you will create additional filters for the webpage. The advanced filtering capability will allow users to filter the data by multiple factors.

Background

The webpage and dynamic table Dana has built are working as intended, but Dana has decided she would like the table to be filtered even further. While it currently filters by the date, she would like to include the ability to filter by more column headers. The ability to pinpoint a search by date and country, for example, would go a long way in providing more in-depth analysis of UFO sightings.

Objectives

The goals of this challenge are for you to:

- Create, update, and deploy JavaScript functions to provide additional table filters.

- Update and deploy `forEach` (`for` loop) to loop through the filters and update them with user input.
- Update and populate the dynamic filters and table using JavaScript and HTML.

Instructions

Include five total filters in the table:

1. Date
2. City
3. State
4. Country
5. Shape

To create these additional filters, keep the following points in mind:

- You will need to create a new function that will replace your `handleClick();` function. This function saves the element, value, and the id of the filter that was changed.
  - Create an `if-else` statement to add filter data from input, or clear the filter if no input data exists.
- Additionally, create a function named `filterTable();` that will perform the following actions:
  - Set the filtered data to the table.
  - Loop through all of the filters and keep any data that matches the filter values.
  - Rebuild the table by calling the `buildTable();` function created earlier.
- Finally, using `d3.selectAll();`, attach an event listener to pick up changes that are made to each filter.

Below are links to starter code to help you build the additional filters. One of the files contains pseudocode, while the other is a blank slate. You can work with either file,

though one provides an additional challenge.

Download the file with pseudocode (pseudoCode.js)

Download the file without pseudocode (starterCode.js)

Submission

Make sure your repo is up to date and includes the following:

- A `README.md` file containing a short description of your project
- The completed code saved in the proper folder structure. **Hint:** Make sure your JavaScript file is saved as `app.js`
- A clean website with a functioning filter table

Submit the link to your repository through Canvas.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 11 Rubric

Criteria	Ratings					Pts
Written Summary	**25.0 pts** **Mastery** Presents a cohesive written summary that describes the purpose of the project, the technologies used, and 1 recommendation for further development.	**18.75 pts** **Approaching Mastery** Presents a cohesive written summary that describes the purpose of the project, the technologies used, and 1 recommendation for further development.	**12.5 pts** **Progressing** Presents a developing written summary that describes the purpose of the project and the technologies used.	**6.25 pts** **Emerging** Presents a limited written summary that describes the purpose of the project and the technologies used.	**0.0 pts** **Incomplete** No submission was received, submission was empty or blank, submission was not published to github pages, or submission contains evidence of academic dishonesty	25.0 pts
Filters	**25.0 pts** **Mastery** Page includes filters for date, city, state, country, and shape, which function without error.	**18.75 pts** **Approaching Mastery** Page includes filters for four of the five categories (date, city, state, country, and shape), which function with no more than two minor errors.	**12.5 pts** **Progressing** Page includes filters for three of the five categories (date, city, state, country, and shape), which function with no more than three minor errors.	**6.25 pts** **Emerging** Page includes filters for two of the five categories (date, city, state, country, and shape), which function with significant errors.	**0.0 pts** **Incomplete** No submission was received, submission was empty or blank, submission was not published to github pages, or submission contains evidence of academic dishonesty	25.0 pts
forEach	**25.0 pts** **Mastery** The forEach function is updated to loop through all five filters and updates based on user input.	**18.75 pts** **Approaching Mastery** The forEach function is updated to loop through four of the five filters and updates based on user input, with one or two minor errors.	**12.5 pts** **Progressing** The forEach function is updated to loop through three of the five filters and updates based on user input, with more than three minor errors.	**6.25 pts** **Emerging** The forEach function is updated to loop through two of the five filters and updates based on user input, with more than three minor errors.	**0.0 pts** **Incomplete** No submission was received, submission was empty or blank, submission was not published to github pages, or submission contains evidence of academic dishonesty	25.0 pts

Criteria	Ratings					Pts
HTML	**25.0 pts** **Mastery** The HTML page displays the filters, table, and article with no errors. The HTML uses 3+ Bootstrap elements to display a cohesive, professional style..	**18.75 pts** **Approaching Mastery** The HTML page displays the filters, table, and article with one or two minor errors. The HTML uses 3+ Bootstrap elements to display a cohesive style.	**12.5 pts** **Progressing** The HTML page displays the two of the following: filters, table, and article, with more than two minor errors. The HTML uses 2+ Bootstrap elements to display a developing style.	**6.25 pts** **Emerging** The HTML page displays one of the following: filters, table, or article, with significant errors. The HTML uses 1+ Bootstrap elements to display a limited style.	**0.0 pts** **Incomplete** No submission was received, submission was empty or blank, submission was not published to github pages, or submission contains evidence of academic dishonesty	25.0 pts
					Total Points: 100.0	

Module 11 Career Connection

Nice job this week. You explored JavaScript—a complex but valuable language to understand, particularly when working on client-side visualization. JavaScript is the go-to-standard for transferring information to consumers via the internet.

A strong foundation in JavaScript can set you apart from other data visualization candidates. It provides technologies that deliver data visualization across the internet (e.g., web hosting). Also, JavaScript opens up a wealth of fun, dynamic libraries to work with, such as **D3.js (https://d3js.org/)** and **Chart.js (https://www.chartjs.org/)**. Of course, JavaScript has its challenges, but nothing this valuable was ever *too* easy.

In this section, you will learn the following:

1. How JavaScript can be used daily in your new position
2. How you can showcase this new skill to be ***Employer Competitive***
3. How to solve common algorithm questions using JavaScript

Employer Competitive Advantage

To get noticed by recruiters, add JavaScript as a skill to your resume. By doing so, you're demonstrating knowledge that extends beyond Python and data analysis, which will set you apart from other candidates. For help developing your resume, see the resources listed at Milestone 4.

Milestone 4: Creating Your Resume
(https://courses.bootcampspot.com/courses/138/assignments/1734)

Technical Interview

Whether you are pursuing a data role or considering a JavaScript development role, the industry expects candidates to demonstrate expertise through algorithms.

The algorithms explored in this section are not typically written on the job; however, they are often used to test coding ability during the interview process.

Preparation

When preparing for a technical interview, keep two things in mind: Just do your best, and demonstrate your thought process.

Just Do Your Best

Remember that potential employers are measuring your performance against other interviewees, and you have no way of knowing who is interviewing for a role. Maybe the other candidates aren't as strong as you. Maybe nobody gets all the answers right. It's hard to know how to perform better in relation to other candidates when you don't know who they are. That frees you up to focus on **yourself** and delivering your best performance when interviewing.

Demonstrate Your Thought Process

The algorithms we are about to explore are quite difficult. They are used in interviews because they are hard. Still, that's no reason to worry. During a technical interview, you're not expected to get every answer right. In fact, interviewers are far more interested in your process. In other words, they want to see how you tackle the problem, not just the solution. In this situation, as in many things in life, it's about the journey, not the destination.

In fact, during the technical interview, the difficulty of the questions continue to increase until you reach a question you're unable to answer. The interviewer wants to observe you under pressure and will focus on your process for finding an answer, not whether you get the right answer.

Technical Interview

Solving JavaScript algorithms during the technical interview is often done as a whiteboarding exercise. You might use a dry-erase marker and a whiteboard and, with a coding prompt, be expected to write out your solution on the board. Sometimes, whiteboarding takes place using a computer or a pen and paper. Still, the steps are the same:

1. Pseudocode
2. Code
3. Optimize

Always write pseudocode for your solution first—don't write any actual code. Use plain English along with certain keywords (e.g., "loop" or "if-statement") to map the logical steps to solve the problem. Pseudocoding tackles two major parts of the coding interview:

1. Syntax

   This brings us back to the whole process of solving a problem again. Interviewers know that developers can forget syntax. We can't remember everything all the time—especially if JavaScript isn't our first language.To be fair, there are a lot of curly braces, brackets, and other weird things in JavaScript. Pseudocoding our solutions allows the interviewer to see which steps we would take, in a logical and methodical approach, knowing that we could easily look up syntax on the internet if this were not an interview.

2. Time Management

   You might be given a huge problem with little time to solve it. Pseudocoding allows us to plug those gaps. Pseudocode first, then start coding your solution. If you don't have time to code out the entire thing, then at least you

have the pseudocode already there to fill those spaces. Only if you manage to code out the entire solution do you then want to look at optimizing it, time-permitting.

So, remember: *p*seudocode, code, then optimize.

Now tackle the following issues using your favorite code editor. Don't use the internet, and give yourself a maximum of 30 minutes to work on the solution. You can complete the solution in either ES5 or ES6.

Challenges

Challenge 1: Array Left Rotation

A left rotation on an array shifts each element of the array one unit to the left. In this challenge, given an array `arr` and a number `n`, perform `n` rotations on the array and return the updated array.

Sample Input

```
arr = [1,2,3,4,5];
```

```
n = 2;
```

Sample Output

```
[3,4,5,1,2]
```

The Code

```
function rotateLeft(arr, n) {
    // YOUR CODE HERE
}
```

The Pseudocode

```
    // create a copy of the array and save it in a temporary variable
    // loop through the array as many times as we need to do rotations
    // remove the first item of the array, store it in a temporary variab

    // add that temporary variable back at the end of the array
    // once completed, return the the newly rotated array
```

The Solution

```
function rotateLeft(arr, n) {
    // create a copy of the array and save it in a temporary variable
    var temp = arr.slice(0);
    // loop through the array as many times as we need to do rotations
    for (var i = 0; i < n; i++) {
        // remove the first item of the array, store it in a temporary va
        var first = temp.shift();
        // add that temporary variable back at the end of the array
        temp.push(first);
    }
    // once completed, return the the newly rotated array
    return temp;
}
```

The Optimized ES6 Solution

```
rotateLeft = (arr, n) => {
    let temp = arr.slice(0);
    for (let i = 0; i < n; i++) {
        let first = temp.shift();
        temp.push(first);
    }
    return temp;
}
```

Challenge 2: CamelCase Converter

Write a function that takes in a string parameter, `str`, and converts any dashed or underscored words into CamelCase. If the first word of the input is capitalized, it should retain its capitalization in the output (i.e., upper CamelCase)

Sample Input

```
toCamelCase("this-is-a-string")
```

```
toCamelCase("This_is_not_an_integer")
```

Sample Output

```
 "thisIsAString"
```

```
"ThisIsNotAnInteger"
```

The Code

```
function toCamelCase(str) {
     // YOUR CODE HERE
}
```

The Pseudocode

```
     // take the input string, and split it where there is an underscore
    // use a loop to go through each item of the splitSentence
    // take the first item in that sentence, capitalize it, and then remo
    // join the array back together

    // return the solved string
```

The Solution

```
function toCamelCase(str) {
     // take the input string, and split it where there is an underscore
    var splitSentence = str.split(/[-_]/);

    // use a loop to go through each item of  the splitSentence
    for (var i = 1; i < splitSentence.length; i++) {
 // take the first item in that sentence, capitalize it, and then remove
        splitSentence[i] = splitSentence[i].charAt(0).toUpperCase() +

    }
    // join the  array back together
    var camelCaseStr = splitSentence.join('');

    // return the solved string
    return camelCaseStr;

}
```

The Optimized ES6 Solution

```
toCamelCase = str => {
    let splitSentence = str.split(/[-_]/);
    for (let i = 1; i < splitSentence.length; i++) {
        splitSentence[i] = splitSentence[i].charAt(0).toUpperCase() +
    }
    const camelCaseStr = splitSentence.join('');
    return camelCaseStr;
}
```

Challenge 3: isPrime

Write a function that takes in an integer argument and returns a Boolean value stating whether the integer is prime. For the purpose of this challenge, a prime

number is considered a natural number greater than 1 that has no positive divisors.

Sample Input

```
console.log(isPrime(1));
console.log(isPrime(29));
console.log(isPrime(30));
```

Sample Output

```
false
true
false
```

The Code

```
function isPrime(integer){
// YOUR CODE HERE
}
```

The Pseudocode

```
        // Check if the number is equal to, or less than, 1. If so, retur

        // Otherwise, start to loop through the numbers that precede the
        // if the integer can be divisible  by [i], return false because

    // Else, return true because the number IS prime
}
```

The Solution

```
function isPrime(integer) {
    // Check if the number is equal to, or less than, 1. If so, return fa
```

```
    if (integer <= 1) {
        return false;
    } else {

        // Otherwise, start to loop through the numbers that precede the
        for (var i = 2; i < integer; i++) {
            // if the integer can be divisible  by [i], return false beca
            if (integer % i === 0) {
                return false;
            }

        }
    }

    // Else, return true because the number IS prime
    return true;
}
```

The Optimized ES6 Solution

```
isPrime = integer => {
    if (integer <= 1) {
        return false;
    } else {
        for (let i = 2; i < integer; i++) {
            if (integer % i === 0) {
                return false;
            }
        }
    }
    return true;
}
```

Continue to Hone Your Skills

If you're interested in learning more about the technical interviewing process and practicing algorithms in a mock interview setting, check out our **upcoming workshops.** **(https://careerservicesonlineevents.splashthat.com/)**