HTML & CSS In 6 Days

Day 1

HTML Fundamentals

HTML Basics.

HTML (Hypertext Markup Language) is the foundation of web development. Hypertext is text that links documents on the internet together through hyperlinks. Markup refers to the tags and indicators used in HTML to indicate things such as headers, images, paragraphs, etc. HTML on its own makes up the skeleton of a webpage.

All HTML elements are made up of tags. For example:

```
<element>content</element>
```

where `<element>` is the opening tag of the given element (such as a paragraph or header) - this indicates the beginning of the element. `content` indicates the element's contents (paragraph text, image source, etc.), and `</element>` indicates the closing tag of the element.

A practical example of this would be a basic button:

```
<button>Buy Now</button>
```

where `<button>` indicates that we are creating a button. `Buy Now` is the text that will be inside the button, and `</button>` indicates that we are done writing the button element.

Building A Basic Web Page

We begin by creating the simplest possible webpage. In a file called `myWebsite.html`, we will write the following:

```
Hello World
```

Now opening this file will show a very simple webpage with our "Hello World" text in the top left corner of the page.

Now we can change this (or write a new line of code below it) to add a button as before:

```
<button>Play</button>
```

You can create a heading with the `h#` tag - where `#` is any whole number 1-6. `h1` is the largest heading type, and each subsequent number becomes incrementally smaller. Everything beyond `h1` is known as a subheading.

```
<h1>This is a heading</h1>
```

Headings are important for search engine optimization, as they indicate which keywords search engines will use to locate your website.

While we were able to print out plain text without tags before with Hello World, it is inadvisable to do so. All elements in an HTML file should be properly tagged. If you want to output regular text on your website, you will use the paragraph tag, indicated with `<p>`.

```
<p>this is some text</p>
```

If you added the button under 'Hello World' at the start of this section, you would note in the actual webpage that they were generated side-by-side rather than each on their own lines. This is because the 'Hello World' text was not tagged, and therefore not handled properly. Using HTML tags in this way ensures that all elements are processed properly. You can test this out for yourself by creating two different elements (such as a paragraph and a button) on different lines in your HTML file, and observing the results.

With all of this knowledge, we can create a very simple web page.

```
<h1>Welcome to my webpage</h1>

<h2>HTML</h2>

<p>HTML stands for Hypertext Markup Language and is used to create web pages.
</p>
<button>Learn More</button>

<h2>CSS</h2>

<p>CSS stands for Cascading Style Sheets and is used to design web pages and
make them responsive.</p>
<button>Learn More</button>

<h2>JavaScript</h2>
```

```
<p>JavaScsript is a programming language used to make web pages dynamic and
interactive.</p>
<button>Learn More</button>
```

Tags and Attributes

As we know with standard elements, there is an opening tag and a closing tag. However, there are certain types of HTML elements that do not require closing tags, as they do not contain any elements. These are called self-closing tags.

One example of a self-closing tag is the line break, which is exactly what it sounds like: it breaks the line of text where it is inserted. The syntax for a line break tag is `<br/>`. Note that there is only one tag, and it ends with the front slash. Example usage for a line break tag looks like this:

```
<p>This is an example <br/> of a line break.<p>
```

Which will look like this on our web page:

```
This is an example
of a line break.
```

Another self-closing tag is the horizontal line tag, denoted by `<hr/>`. This adds a single horizontal line across the screen to visually break up the page.

Another self-closing tag is `<input/>` . This is a complex tag that will require more in-depth discussion later, but its basic purpose is to gather user input. There can be many different types of input, such as text input, numbers, time, color, file upload, etc. The way this input type is determined is with attributes.

Attributes are defined within the opening tag of an element. Since input is a self-closing tag, the attribute goes before the slash. Let's say we want to get a number from the user. Our input tag and attribute would look like this:

```
<input type="number"/>
```

Note that there is a space after the tag definition, followed by `type=` and the attribute in quotation marks. Note also that there are no spaces in the attribute definition.

Tags can have multiple attributes. Another useful attribute for the input tag is `placeholder`, which inserts a placeholder value into your input field that will be replaced by your actual user input. To amend our current example, it would look like this:

```
<input type="number" placeholder="Enter your age"/>
```

In this case, there *are* spaces between the different attributes, and of course there are spaces in the string that we will input into the placeholder field. Otherwise, there is no delineation between the two attributes.

The Basic HTML Structure

The filename of `index.html` is significant in HTML. `index.html` serves as the homepage of your website - anytime the domain name of your website is entered, this is the first page that will appear. We will create a new `index.html` file and use it for the duration of this lesson.

Now let's talk about structure and high-level syntax. When starting a new HTML file, you will create new opening and closing tags with the `<html>` element. All other code for the site will be between these tags. Once you've inserted the `</html>` tag, the file should be complete. No further elements should go beyond this closing tag.

Similarly, most HTML pages have opening and closing tags for `<head>` and `<body>`. These go within the `<html>` tags, meaning nested elements are possible. A nested element is when an element exists within another element. In this case, `<head>` and `<body>` are nested within `<html>`. The proper terminology for a nested element is a child element. `<head>` and `<body>` are child elements of `<html>`. Conversely, `<html>` is the parent element of `<head>` and `<body>`.

Your `<head>` element contains all of your settings and other resources to be imported from the internet, such as fonts, icons, and similar items. More on those later. One useful tag you can insert into the `<head>` element is `<title>`. This is the name of the page as it appears in your browser tab. Meaning you could have HTML that looks like the following, and it will display "YouTube" in the browser tab.

```
<html>
    <head>
        <title>YouTube</title>
    </head>
</html>
```

Note also that child elements are tabbed into their respective parent elements. This is not technically necessary in order for the HTML to be properly loaded, but it is best practice in order to maintain readability.

While the head of the HTML code determines the settings and important info, the body will contain all of the visual data. This is the part of the code we were altering in the previous

sections. All of our headers, paragraphs, buttons, etc. will go in the `<body>` element of our code. As such, if you want a very basic web page that displays its name as 'Cool Website' and contains a title and a paragraph, your HTML would look like this:

```
<html>
    <head>
        <title>Cool Website</title>
    </head>
    <body>
        <h1>Welcome to my cool website!</h1>
        <p>I'm so glad you decided to visit my little website :)</p>
    </body>
</html>
```

In Visual Studio Code, you can insert the basic HTML structure into your document with a single command. In an empty document, simply type `!` into the coding field, and press enter. The resulting code will be slightly more complex than what we've learned so far, but it functions the same. Let's take a look at it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

</body>
</html>
```

- `<!DOCTYPE html>` - This line tells the browser that we are using HTML, specifically HTML 5. Previous versions of HTML would require a longer and more complex line of code to specify which version is being used.
- `<html lang="en">` - as we know, this is the opening tag of our webpage. This has the "lang" attribute, which specifies what language our page is in. In our case, the language is English.
- `<head>` - the start of our header; the place where we insert all of the settings for our webpage. Note here that it is not indented. This is not a dealbreaker, as it would only be indented into the HTML tag, which is all-encompassing and could otherwise be a pain to manage.

- `<meta charset="UTF-8">` - This is a meta tag defining the character set to be used on our page. A character set is essentially the list of letters, numbers, and symbols that are available for the webpage itself. We are specifying UTF-8, as it is the most common character set. This tag isn't typically needed by most modern browsers, but it is still best practice to include it to accommodate older browsers such as Internet Explorer, especially if your webpage is in a language other than English.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">` - This tag adjusts the viewport (or visible area of the webpage) to be the same width as the device accessing the webpage. This is useful to accommodate mobile phones, as their viewport will be significantly smaller than that of a standard computer/laptop screen.
- `<title>Document</title>` - This is our title tag. The contents of which can (and should) be changed to better suit the webpage you are building.

The rest of this framework is elements that we already know. There is the section for the body, which contains a blank line on which you can begin writing your code, and then the closing tags for the body and the HTML file as a whole.

Add Images To Your Website

Images in HTML are handled with the `<img/>` tag, which is self-closing. This tag must be accompanied by the `src=""` tag in order to specify the image you wish to use. If your image is not located in the same directory as your HTML file, you must insert the file path into the quotation marks. If it is located in the same directory, you can simply input the file's name and extension.

For example, I am using my web safe SGB V2 profile picture for practice. It is located in the same directory as the `index.html` file that I am editing. Therefore, if I want to insert this image into my HTML file, it will look like this:

```
<img src="SGBv2_websafe.png"/>
```

Note that best practice is to have a separate folder for your images when creating a webpage. I will be creating an `images` subfolder within my project directory, and place my image in there. As such, I will need to update the image source attribute to include the `images` directory.

```
<img src="images/SGBv2_websafe.png"/>
```

Relative Paths

As mentioned above, when accessing resources such as images, videos, icons, etc. you will need to specify the file path where the desired resource is located. "Relative Paths" refers to

the way HTML automatically interprets the file path you input. You don't need to input the entire file path all the way from the root of your hard drive every time - HTML interprets the start of the path as the root of the project folder where your `index.html` file is located, just as above where we only needed to specify `images/(file)` rather than `C:/users/username/Documents...` etc.

If, for example, our image file was one file above our `index.html` file, we could navigate to it via `../`. This indicates that we want to move one folder up from the current position of our `index.html` file. This implementation would look like the following:

```
<img src="../SGBv2_websafe.png"/>
```

You can add `../` to navigate "up" one folder as many times as you need to. If your `index.html` file was two subfolders deeper than your image file, you could input the following:

```
<img src="../../SGBv2_websafe.png"/>
```

Understanding Image Attributes

Additional attributes can be added to images in order to improve SEO and make your code more understandable. The `alt=""` attribute allows you to add alternative text to describe your image. This gives the image keywords that search engines can use to find your image, and helps to describe the image with words rather than requiring you to look for the referenced image each time you go to edit your code. This alt text is also displayed if there is an error when displaying your image, making it easier to diagnose issues if needed.

For example, with my SGB image, I might include an `alt` attribute that looks like this:

```
<img src="images/SGBv2_websafe.png" alt="SuperGiggleBot profile image"/>
```

In addition, alt text is used with visual screen readers for blind users so they can better understand how your web page is laid out.

Another useful pair of attributes for images is `height` and `width`. This sets your image's respective height or width in pixels according to the number you input for these attributes. If you only set one of these attributes, the image will scale accordingly to avoid squash/stretch. If you set both, this safeguard is overridden, and squashing and stretching may occur if your input values are different from the aspect ratio of your image.

It is important to set your image size in HTML even though it can be done in CSS as well. This is because images load slower than text. If you don't specify your image size in advance, your browser will not allocate enough space for it on your page as it is loading, and can result in a

poorly rendered page. Setting your image size in HTML rather than CSS allows the browser to pre-allocate that space so it can go where it needs to once the page loads fully.

Semantic HTML Tags

Writing semantic HTML means using HTML elements to structure your content based on each element's meaning, not its appearance. For example, an `<h1>` element in HTML is considered the most important heading, because it is the first and largest. Search engines will prioritize the keywords used in these headings due to their relative importance. While it is possible to make a paragraph appear the same size as a heading, it is important to use a heading for important keywords for SEO purposes. The visual appearance of the text can be changed in CSS later.

Text Formatting Tags

Here is a quick rundown of tags to be used nested within paragraph elements to change their formatting.

- `<strong>` - This displays the text as bolded, and increases the semantic meaning of the text within this element.
- `<b>` - This displays the text as bolded, but does not increase the semantic meaning of the text within this element.
- `<em>` - This displays the text as italic, and increases the semantic meaning of the text within this element.
- `<i>` - This displays the text as italic, but does not increase the semantic meaning of the text within this element.
- `<mark>` - This displays the text as highlighted, and is used to indicate text of special relevance.
- `<del>` - This represents text that was removed from the displayed document. It is displayed as strikethrough text.
- `<ins>` - This represents inserted text. Meant to signify text that has been added after the page's initial creation. This is displayed as underlined text.

Note that while these tags do change the style of the text being formatted, HTML is not inherently intended for text styling. These tags are better used for their semantic meaning. CSS can be used later to style your text as desired.

Add Hyperlinks To Your Website

Hyperlinks allow you to add links to other websites, as well as other pages within your own site. This element is the single most important tag you can use in HTML, as the entire internet is made up of hyperlinks. You can insert the hyperlink element into your HTML using the `<a>` tag, which stands for "anchor." Usage is as follows:

```
<a href="https://www.youtube.com/example">This is a link to YouTube</a>
```

The `href` attribute is your hyperlink reference - the actual link where your anchor element will go when clicked. Note that the full URL is required here, including the `HTTPS://` all the way to the `.com` (or relevant domain extension therein)
The text within your anchor element is what text will be displayed on the webpage where this hyperlink exists.

Anchor tags can also be used to link other pages on your site or additional HTML elements within your own code. For example, let's say you have a second file in your website project called `about_us.html` which contains the "About Us" page. To link to this page, you would include the relative path to `about_us.html` in the `href` tag rather than a URL. See above about relative paths.

Anchor tags have another important attribute: `target=""`. This tag indicates where the link will open when clicked. By default the target attribute should be set to `target="_self"` - indicating that when the link is clicked, it will be opened in the same tab as our current webpage. Setting this attribute to `_blank` will open the link in a new tab.

Creating Bookmark Links

A bookmark link is an HTML element within the same page that you are currently viewing. Think about Wikipedia articles - in the left panel there are several links to the materials inside the table of contents on that article. Clicking one of these links will take you to another part of the same page.

In order to create a bookmark link, you first require a page that requires scrolling down to view more contents. You can mock this up by creating a paragraph element with a bunch of random words. In Visual Studio code, this can be achieved with the `Lorem100` keyword, and then pressing enter. It will generate 100 random words of Lorem Ipsum text that you can copy/paste several times to fill out your web page.

In the middle of your multiple copies of lorem ipsum, you can insert a heading that will serve as our bookmark link. So far, your code should look like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Homepage</title>
</head>
<body>
```

```
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <h2>Hello World</h2>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
</body>
</html>
```

Now in your h2, you can set an attribute called `id=""`. This ID can be any name you want it to be, but you have to ensure there are no spaces in it. Use dashes or camel case as needed. For this example, we will call it `my-example-heading`

```
<h2 id="my-example-heading">Hello World</h2>
```

Now using this ID, we can reference it with a hash in an anchor element to create a bookmark link.

```
<a href="#my-example-heading">Bookmark Link</a>
```

Now clicking the `Bookmark Link` on our webpage will automatically scroll us down to the example heading.

Here is what that code looks like altogether:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Example Webpage</h1>
    <a href="#my-example-heading">Bookmark Link</a>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <h2 id="my-example-heading">Hello World</h2>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
    <p>pretend this paragraph contains lots of text</p>
```

```
</body>
</html>
```

Lists

HTML has two types of lists: Ordered and Unordered. An ordered list is a numbered list, and an unordered list utilizes bullet points.

With an Ordered list, you will begin by defining the `<ol>` tag, short for 'ordered list.' Note that this tag is not self-closing. Nested within the `<ol>` tag is each item, tagged as `<li>`. For example, an ordered list with three entries would look like this:

```
<ol>
    <li>Item 1.</li>
    <li>Item 2.</li>
    <li>Item 3.</li>
</ol>
```

Unordered lists are made almost exactly the same way, with the only difference being that you define them with the `<ul>` tag as opposed to `<ol>`. Otherwise you still categorize each item nested within the `<ul>` element with `<li>`, and close the tags where appropriate.

You can also nest lists within lists.

```
<ul>
    <li>Here is an item</li>
    <li>Here is another item</li>
        <ol>
            <li>sublist item 1</li>
            <li>sublist item 2</li>
        </ol>
    <li>Now we are back to the main list</li>
</ul>
```

And this is what it will look like on your webpage:

- Here is an item
- Here is another item
    1. sublist item 1
    2. sublist item 2
- Now we are back to the main list

Comments

Comments in any coding language are important for the sake of leaving notes for yourself or other developers, as well as temporarily marking a section of code to be ignored by your browser. Anything in your code that is defined as a comment will not be run as code, but will still be visible to you in the IDE.

To leave a comment in HTML, you use the following markers: `<!--` and `-->`
These markers go at the beginning and end of your comment, and everything between them will be marked as a comment. For example:

```
<!-- This is an example of a comment. It can go on for as long as I want it
to, and nothing within the specially-defined markers will be interpreted as
HTML by the browser. Once I am done writing my comment, I will use the closing
marker. -->
```

Advanced HTML

Forms

Forms and their input fields allow you to collect data from the user, and even provide a response on the webpage if needed. Forms are used for search bars, file uploads, comments, and much more. Actually processing the input gathered from a form is done via another language such as JavaScript or PHP, but HTML allows for the form type to be used on the page.

Forms in their most basic state are created using the `<form>` tag. Nested within this tag should be the `<input/>` self-closing tag. It is important to use the `type=""` attribute in order to define what kind of input you wish to gather: text, date, number, etc.

Above this input tag, you can utilize the `<label>` tag to define to the user what this form is intended for. For example, if we want to ask the user for their name, it might look like this:

```
<form>
    <label for="username">Enter Your Name:</label>
    <input id="username" type="text" placeholder="Johnny Appleseed"/>
</form>
```

Recall that the `placeholder` attribute will give example text in your form, to be replaced by the actual text input by the user.

Note additionally that we are introducing a new attribute in the `label` tag: `for=""`
This tag serves to link the label to the input field, allowing for processing with PHP or JavaScript later. This is important to learn now to create effective websites. We also need to define our input field with the appropriate `id` attribute to link to our label.

When I tested this out myself, I first created the above webpage without linking the label and form using `for` and `id` attributes. It was treated by my browser as a standard text field. When I then linked them under the `username` ID, I was prompted with autofill in my browser to input previous usernames that I have used on other websites. Bear this in mind when creating login pages or just using forms in general.

In addition, linking labels and forms allows users to click on the label, and their typing cursor will automatically be placed within the text field, allowing greater ease-of-use on your webpage. This is especially useful when utilizing the checkbox type input field.

Another useful input type is the `<select>` tag. This is used to create dropdowns with multiple options. Options are defined as `<option>` nested within `<select>` tags. See below:

```
<select>
    <option>Option 1</option>
    <option>Option 2</option>
    <option>Option 3</option>
</select>
```

Each option can (and should) be defined with a `value=""` attribute for further processing in JavaScript. The names within the tags are simply text to better assist the user. The `value` attribute actually defines what each option's value is in code.

With dropdown options, you can also set an attribute called `selected`, which will be the default selection when the dropdown menu is initially loaded on your page. There is nothing else to define with this attribute - it is not `selected=""` - it is simply `selected`.

With this info in mind, if I wanted to create a dropdown on my webpage that had option 2 as the default when the user first navigates to the page, my HTML would look something like this:

```
<form>
    <label for="dropdown">Select An Option</label>
    <select id="dropdown">
        <option value="option1">Option 1</option>
        <option selected value="option2">Option 2</option>
        <option value="option3">Option 3</option>
    </select>
</form>
```

Input Types

As discussed previously, there are many different input types in HTML. This section aims to go over the most common ones. All input types can be referenced here

- `<input type="text"/>` - this input type allows for the general input of text. The `placeholder=""` attribute can be used here.
- `<input type="number"/>` - this is specifically for numbers. An up/down arrow will appear on the right side of this input box to increment your input amount by 1. You can set the `value=""` attribute to set a default number to go here.
- `<input type="checkbox"/>` - this will create a single checkbox. Best paired with the `<label>` tag, and possibly a list.
- `<input type="password"/>` - this is similar to a standard text input, but all of the input characters will be obscured by dots. Useful for login pages.
- `<input type="email"/>` - this also functions similarly to a text input field, except the browser automatically recognizes that it is an email field and will check for validity/autofill as required.
- `<input type="submit"/>` - this creates a button to submit the contents of all filled-out input fields on the page. This will send the info from these fields for processing via JavaScript or PHP.
- `<input type="reset"/>` - this creates a button that will clear all input fields on the page without submitting their contents.
- `<input type="radio"/>` - this is a selection type similar to a check box. Bind multiple to the same group with the `name=""` attribute, and the user can only select one option from each group. Recommended to pair with a `<label>` tag for clarity.
- `<input type="date"/>` - creates a calendar input field to select a date.
- `<input type="time"/>` - creates a clock input field to select a time.
- `<input type="color"/>` - creates a color picker to select a color/hex code.
- `<input type="file"/>` - allows a user to select a file from their computer to upload to the website.
- `<input type="search"/>` - Functionally similar to a text input field, but contains an X icon to the right that allows clearing the search/text.

Tables

A table in HTML is a way to organize data on your webpage via rows and columns. The rows are created by you, the developer, and the columns are created automatically by the browser. You can create a table using the `<table>` tag.

Nested within the `<table>` tag is each row tag, indicated by `<tr>` (table row). The first row is typically the header row, which is used for the names of the data entries to go in each column. This header is indicated by `<th>` (table header), and is nested within the first `<tr>` tag.

Let's begin creating a simple table to make the layout a bit more clear:

```
<table>
    <tr>
        <th>firstname</th>
        <th>lastname</th>
        <th>gender</th>
        <th>year of birth</th>
    </tr>
</table>
```

Now you can see that we've created a table. It contains one header row, and that row will be split into four columns due to the four header entries on the first row. We can then add another row to this table by adding another `<tr>` tag, with `<td>` (table data) as our tag to indicate data.

```
<table>
    <tr>
        <th>firstname</th>
        <th>lastname</th>
        <th>gender</th>
        <th>year of birth</th>
    </tr>
    <tr>
        <td>Albert</td>
        <td>Einstein</td>
        <td>Male</td>
        <td>1879</td>
    </tr>
</table>
```

Now we have added data to our table. Each `<td>` element is going to be a column in our table. We can create more rows by adding more `<tr>` elements with `<td>` elements nested within. Semantically, it is also important to differentiate the table head from the table body by using the `<thead>` and `<tbody>` tags respectively. Let's add one more entry to our table, and wrap the head and body into its respective tags for semantic relevance.

```
<table>
    <thead>
        <tr>
            <th>firstname</th>
            <th>lastname</th>
            <th>gender</th>
            <th>year of birth</th>
        </tr>
```

```
    </thead>
    <tbody>
        <tr>
            <td>Albert</td>
            <td>Einstein</td>
            <td>Male</td>
            <td>1879</td>
        </tr>
        <tr>
            <td>Stephen</td>
            <td>Hawking</td>
            <td>Male</td>
            <td>1942</td>
        </tr>
    </tbody>
</table>
```

Note that despite the differentiation between head and body within a table, *the entire table element will exist within the body element of your larger HTML document.*

Creating Layouts With Tables

In most cases, page layout will be done via CSS. However, sometimes doing some layout via HTML can be useful. For example, by default in HTML, all elements are stacked vertically. However, by using a table, you can arrange some objects horizontally instead.

An example of this would be placing an image and its description side-by-side. Typically if you were to use a standard image and paragraph/header element, they would stack vertically. But you can use a table to make them appear side-by-side on the webpage.

This can be accomplished by placing our image, header, paragraph, and anchor elements within a two-column table. The code to which would look like this:

```
<table>
    <tr>
        <td>
            <img width="500" src="images/chicago.jpg"/>
        </td>
        <td>
            <h2>Cloud Gate, Chicago</h2>
            <p>Cloud Gate, designed by Anish Kapoor.</p>
            <a href="https://en.wikipedia.org/wiki/Cloud_Gate">Learn More</a>
        </td>
    </tr>
</table>
```

Note that I don't need a table header here, I am simply using the table element for alignment.

An important attribute to learn here is the `cellspacing=""` attribute to be used within the `<table>` tag. The number input into this attribute determines the amount of space between cells to be used in pixels. As of right now, the above HTML would make the text appear right up against the image. However, if we adjust our `<table>` tag to include `<table cellspacing="20">`, then we can have a cushion of 20 pixels between the image and text.

Add Videos To Your Webpage

This section is about how to add a video to your webpage from your computer (as you would an image) and not video embeds. See the next section for embedding a video from YouTube.

Videos can be added to your webpage with multiple options. Some are purely decorative and cannot be paused, fast forwarded, etc. And others can include these options. The most basic form of adding a video to a webpage is by using the `<video>` tag. You will also use the `src=""` attribute to point to the location of your video in the same way you would an image. For example, if I have a subfolder in my project file called "videos", and I want to include a video there called "airplane.mp4", it would look like this:

```
<video src="videos/airplane.mp4"></video>
```

Note that unlike images, Video tags are not self-closing. More on this later.

You can also set the height and width similar to that of an image, with the exception that if you specify a different aspect ratio to your original video, it will not squash the video. It will simply display a white border over the excess area of the video.

An attribute unique to videos is `controls`. This does not require further arguments - it simply enables standard video controls such as play/pause, fullscreen, mute, etc.
You can also set the `autoplay` attribute for your video so it will play as soon as the page is loaded. Remove the `controls` attribute and you've got a completely decorative video. It is worth mentioning that autoplay is intentionally blocked by many browsers to prevent user spam. The best way around this is to ensure your video has no audio by including the `muted` attribute. Once again, this requires no further arguments. Finally, you can ensure the video plays on loop by including the `loop` attribute. Otherwise the video will pause on the final frame after its first playthrough.

For clarity, if you want a purely decorative video on your webpage that doesn't have user controls, plays as soon as the page loads, and loops when it's finished, it might look like this:

```
<video src="videos/airplane.mp4" autoplay muted loop></video>
```

Embedding YouTube Videos

To embed a video from YouTube onto your web page, you can go to the desired video on YouTube, select the "Share" option, then select "Embed." This will give you a few options to tweak, such as what timestamp to begin the video on, whether or not controls should be included, etc. Once done, you can copy the CSS by clicking "copy" in the bottom right corner, or highlighting the HTML code in the top right of the window. From here you can return to your code and paste the HTML you just copied in the desired location.

Note that this is not available on every YouTube video, and the option to embed is at the discretion of the uploader.

Define Tooltips

A tooltip is the little window that appears whenever you hover your mouse cursor over an element in a webpage, such as links and images. These tooltips provide info on what the selected element does.

Adding a tooltip is as simple as adding the `title=""` attribute to your desired tag. This functions similarly to alt text, where whatever you put between the quotes will be the tooltip that appears when your mouse cursor hovers over that element. This also improves SEO, as it allows you to add more keywords to your webpage.

For example, to add a helpful tooltip to a button, you might do this:

```
<button title="This is a button">Click Here</button>
```

Intro to CSS

CSS Syntax

CSS, or Cascading Style Sheets, is the foundation for design and style in front-end web development. While HTML is used for inserting the critical components of a webpage, CSS is used to create an engaging appearance and user experience for your website.

At its core, CSS allows you to write rules that dictate how your HTML elements should appear on your website. These rules are always syntactically structured the same way: There is a selector, and a declaration block. For example:

```
button{
    color: white;
}
```

In this example, our selector is `button`, where we are selecting which HTML object we are modifying. Then our declaration block is everything inside of the curly braces `{}`. In the above example, we are only setting the button color to white, but this declaration block can contain multiple instructions. Each set of instructions should follow these rules:

- Each set of instructions should sit on its own line of code.
- All instructions contained within a given declaration block should be indented below its respective selector.
- Each set of instructions should begin with the parameter being modified, followed by a colon. In the example above, it is `color:`.
- Each set of instructions should end with a semicolon `;`.

So for example, if we were modifying a paragraph with more than one set of instructions, it would look like this:

```
p{
    color: black;
    font-size: 14pt;
    font-family: garamond;
}
```

Note: More time will be spent later on individual properties that can be modified with CSS, and how their parameters work.

Selectors in CSS are a bit more complex - in the above two examples, we are selecting and modifying *all* elements of that type. So all buttons on the page would be white, and all paragraphs on the page would have black, 14pt, Garamond text. Selecting specific elements, or all elements that fit certain parameters can become a bit more complex. See below for an example of this:

```
header > div > h1 + button:hover{
    background-color: yellow;
}
```

I will not be explaining what this does right now, this is more of an example demonstrating how CSS largely depends on the logic of selectors.

Implementing CSS Into Your HTML Code

There are three ways to implement CSS styling into your HTML code:

1. Inline CSS

2. Internal CSS
3. External CSS

Let's say we have a basic index.html file containing a simple heading:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h2>This is a heading</h2>
</body>
</html>
```

We will also establish for the sake of tutorial that our goal is to turn this header blue.

With inline CSS, we access CSS properties using the `style=""` tag on our desired element. Using this attribute allows us to modify any parameter that would be available to us in CSS. In this example, we would use this attribute on our `<h2>` tag, which would look like this:

```
<h2 style="color: blue;">This is a heading</h2>
```

If we wanted to change multiple properties with inline CSS, we don't need to use the `style` attribute again, we can simply define more parameters in between semicolons. For example, if we were to also set the background color to yellow, it would look like this:

```
<h2 style="color: blue;background-color: yellow;">
    This is a heading
</h2>
```

Inline CSS is useful for very minimal adjustments on specific elements in your HTML code, but it is not recommended for larger stylistic changes, as the code can become messy and difficult to read. For larger projects, it's recommended to either use internal or external CSS.

With internal CSS, you will create a new kind of HTML tag in the *header* section of your `.html` file. This new tag is `<style>`, and contains all of the CSS code between its tags. For example, if we once again wanted to set the header text to blue and background color to yellow using internal CSS, it would look like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        h2{
            color: blue;
            background-color: yellow;
        }
    </style>
</head>
<body>
    <h2>This is a heading</h2>
</body>
</html>
```

Note here that as the final tag in the `<head>` section, we've established our CSS styling parameters within `<style>` . In this case, it will affect all h2 elements on this page. This can be overridden on the individual level with inline CSS. So if we had multiple h2 elements, but only wanted one to be different, we could set its own style using inline CSS, which will take precedence over what is established in the `<style>` tag.

Finally there is external CSS, which is the most commonly used by professionals. External CSS involves creating a separate file specifically for CSS, and then linking it to your HTML file. The standard name for this external file is `style.css`

Once you have created your `style.css` file, you must link it to your HTML file by including the following tag in the HTML header section:

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

The two attributes in this `<link>` tag are important: `rel` is short for "relation", and is used to define what kind of file you are linking to your HTML. Other files such as icons can also be linked to HTML, so it is important to include this attribute. Then `href` is used to establish the relative filepath of your CSS file.

To turn our heading text blue and background yellow using external CSS, it would involve two files saved in the same folder. These files will be named and structured as below:

`index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h2>This is a heading</h2>
</body>
</html>
```

`style.css`

```
h2{
    color: blue;
    background-color: yellow;
}
```

Colors

Colors are one of the most important topics in web development, and especially CSS. Until now, we have used named colors in CSS, such as `blue` and `yellow`. CSS offers over 140 named colors for use, and they can be referenced in the documentation. These can be useful when mocking up a website, or if specifics aren't an issue. However, if you require very specific shades of each color for the sake of branding or style, you can input the RGB (red, green, blue) values of the color you would like to use. This can be done with the following syntax:

```
body{
    background-color: rgb(128, 128, 128);
}
```

Note here that the `body` selector is a placeholder for any selector. The lesson to learn is in the color declaration

After specifying what kind of color to change (color, background color, foreground color, etc.) we can use `rgb()` to set up our RGB value. The numbers will always go in order of red, green, then blue. Each of these values can be a whole number anywhere in the range of 0-255. The higher the value, the more intense the selected color.

This RGB parameter allows for the use of over 16 million colors. As such, it is useful to have a color picker on hand that we can use as a reference, and copy/paste the values of our desired

colors.

In addition, it is also possible to use the hex codes for colors instead of names or RGB values. This can also be obtained via the color picker above. Syntax for hex colors looks like this:

```
body{
    background-color: #ff5733;
}
```

Note here that we only need to preface with the hash symbol, and then the hex code of our desired color.

There are other color methods in CSS such as HSL and HWB, but those are not necessary for now. The three discussed here are the most commonly used.

Classes And IDs

Classes and IDs refer to different selectors that can be used in CSS. For the sake of this section, let's say we have an HTML file that contains four paragraphs that have been generally stylized somewhat. We will use Internal CSS for ease of access.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        p{
            background-color: aqua;
            color: blueviolet;
        }
    </style>
</head>
<body>
    <p>This is a paragraph</p>
    <p>This is a paragraph</p>
    <p>This is a paragraph</p>
    <p>This is a paragraph</p>
</body>
</html>
```

If we want to stylize a single paragraph without affecting the others, we can set up an ID. For example, if we only wanted to stylize the second paragraph without affecting any of the others,

we can give it an `id=""` attribute as discussed in the HTML section.

`<p id="redparagraph">This is a paragraph</p>`

now back in the CSS section of our code, we can select just this paragraph by using its unique ID:

```
<style>
    #redparagraph{
        color: red;
    }
</style>
```

Note that when selecting an ID, we preface it with a hash `#` .

Just as with internal vs inline CSS, the selected ID will take precedence in this situation. This is because the ID has a higher specificity than the general paragraph styling that we did before.

We can also use classes to specify certain elements to be changed with CSS. Similar to an ID, we first need to add an attribute to any tag that should be part of this tag. In this case, let's say the fourth paragraph in our test code has the attribute `class="greenelement"` .

To specify a class in CSS, you'll preface it with a period rather than a hash. See below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        p{
            background-color: aqua;
            color: blueviolet;
        }
        .greenelement{
            color: green;
        }
    </style>
</head>
<body>
    <p>This is a paragraph</p>
    <p>This is a paragraph</p>
    <p>This is a paragraph</p>
    <p class="greenelement">This is a paragraph</p>
```

```
</body>
</html>
```

Once again the class will take precedence because it has a higher specificity.

The difference between a class and an ID is in its use case. If you are using an ID, only one element in your HTML code should carry that ID. There can be other IDs in your code, but they cannot have the same name. Classes, on the other hand, can contain multiple elements, and all of them will be styled accordingly.

A single element can be part of multiple classes. The only delineator when specifying these classes in your HTML tags is a space. for example, we can add our fourth paragraph to a second class by simply specifying another class within the same set of quotation marks:

```
<p class="greenelement greybackground">
    This is a paragraph
</p>
```

Now this paragraph belongs to both the `greenelement` and `greybackground` classes.

If you have an element that is both part of a class AND has a unique ID, the ID will take precedence over the class due to its higher specificity.

Day 2

CSS Basics

Control Height and Width of Elements

In HTML, the default width of an element is equal to the width of the screen, while the height is equal to the amount of space its content takes up. Let's say we have a div element (which is a container to hold multiple elements in HTML)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Height and Width</title>
    <style>
        div{
            background-color: blue;
            color: white;
        }
```

```
    </style>
</head>
<body>
    <div>
        <p>Paragraph Text</p>
        <p>Paragraph Text</p>
        <p>Paragraph Text</p>
        <p>Paragraph Text</p>
    </div>
</body>
</html>
```

In the code above, we have a div containing four paragraphs. The div is styled such that the background is blue and the text color of the paragraphs is white. As we edit this HTML, you'll notice that the blue background stretches across the entire width of the browser window. The height of the blue background changes depending on how many paragraphs are contained within the div.

We can manually set the size of this div in CSS.

```
div{
    width: 400px;
    height: 400px;
}
```

Note that we must specify the unit of measurement when setting these values. There are many different units of measurement in CSS such as points, picas, inches, etc. But the most common is pixels, abbreviated as `px`.

Combined with the previously existing CSS to adjust the colors as before, we can now see that the blue background is a square on the screen exactly 400x400 pixels. This will no longer change dynamically as we add or subtract content from this div. Any content that takes up more space than this div's size will simply be cropped out.

If we want to adjust the size of an object relative to the screen size rather than hard coded width and height, we would measure in percentages rather than pixels. Percentages are scaled relative to the object's parent. In our case, our div's parent object is `<body>`. If we wanted our div to always be exactly half the size of our document body, we would establish the following:

```
div{
    width: 50%;
    height; 50%;
}
```

Note that when we do this, the width is visibly half the size of the web page, but the height appears to be unchanged. This is because our document body doesn't have a specified height. Since percentages scale with the element's parent object, we have nothing to scale the height against. We can set the body's max height to something like 500 px, and notice that our CSS now makes sense.

```
body{
    height: 500px;
}
div{
    background-color: blue;
    color: white;
    width: 50%;
    height: 50%;
}
```

Add Borders To Elements

Borders in CSS are made up of three elements: `border-width`, `border-style`, and `border-color`.

For the purposes of this lesson, let's assume we have an HTML document with an otherwise blank div inside the body area. The CSS to modify its border would look something like this:

```
div{
    background-color: blue;
    height: 400px;
    width: 400px;
    border-width: 10px;
    border-style: solid;
    border-color: tomato;
}
```

The first three properties aren't important, but it will help with visibility if you decide to copy this code for yourself.

The `border-width` and `border-color` properties are fairly self-explanatory: these properties define the size and color of your border. Just as with other size-related properties, you must specify your unit of measurement in the `border-width` property. Just as with any color-related properties, you can use named, RGB, Hex, etc. for the `border-color` property.

`border-style` refers to the display of the border itself. In our example above, we have set it to solid, meaning it will be a single continuous line across the entire perimeter of our div. However,

there are other options such as dotted, dashed, ridged, inset, etc. If this property is not defined, your border will not be displayed, even if you have defined the other two properties.

Of course it would be rather frustrating if all three of these properties had to be individually defined each time you wanted to create a border. Fortunately, CSS has shorthand for border creation, where you can set the width, style, and color all on the same line of code.

```
div{
    border: 10px solid black;
}
```

Note the order and syntax: it will always go in order of size, style, color. Also there is no delineation between the three properties aside from a space. No comma separation or anything else. This will improve the speed and efficiency with which you create borders.

Margins

A margin is the space outside the border of an element. It separates this element from any neighboring elements, creating space around it.

Let's say we have an HTML file containing three divs, each with the class `box`. There is some basic styling for the general div class in order to make them visible:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Margins</title>
    <style>
        .box{
            background-color: lightgreen;
            height: 50px;
        }
    </style>
</head>
<body>
    <div class="box"></div>
    <div class="box"></div>
    <div class="box"></div>
</body>
</html>
```

As it stands right now, it appears that there is only one box on screen, despite the fact that there are three. This is because all three are stacked on top of each other. You can't see where one ends and the next begins. To give them each space, we can add margins in our CSS:

```
.box{
    background-color: lightgreen;
    height: 50px;
    margin: 20px;
}
```

Now you can see three clearly separated rectangles. Note also that they are pushed away from the left edge of the screen. Margins give padding on all sides of the element. Much like with borders, you can set the margins for specific sides by specifying `margin-top`, `margin-left`, `margin-bottom`, and `margin-left`. You can also specify all four of these on a single line by specifying four different measurements in your `margin` property. The order will always go top, left, bottom, right. For example, we could set something like the following to customize each side of our margins:

```
box{
    margin: 20px 30px 50px 30px;
}
```

There is also shorthand for only two values, where you can set top/bottom simultaneously, as well as left/right sides. This is similar to the above, but only using two values instead of four.

In CSS, the body contains some padding by default. If you want to set this yourself or otherwise remove it, you can do so by selecting the body and setting the margin to 0. You don't need to specify a unit when setting a value to 0; zero is zero no matter what measurement you're using.

The Box Model

Just as margins add space to the outside of an element, padding will add space to the inside of an element, pushing its contents closer to the center.

Our example HTML for this section is a div with class `box`, lightly stylized for visibility, a 20px margin, and some text on the inside.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Margin and Padding</title>
```

```
    <style>
        .box{
            background-color: lightgreen;
            height: 50px;
            width: 300px;
            margin: 20px;
        }
    </style>
</head>
<body>
    <div class="box">
        This is a box with padding
    </div>
</body>
</html>
```

We can add padding to our box by adding the following to our CSS for the box class:

```
padding: 20px;
```

Once this is saved, two things will happen: First, our text gets pushed closer to the center of the box by 20 pixels. Second, our box gets bigger. The reason for both of these things is that we have essentially added 20 additional pixels to each side of the inside of our box. This works similarly to margins, only padding gets added to the inside rather than pushing content away from the outside. Just as with margins, you can set padding specific to each side similarly to how it is done with margins.

So why does our box get larger with padding despite having a set size? This is due to a CSS concept called the box model. The box model is essentially a CSS box that wraps around every HTML element. It consists of content, padding, borders, and margins. You can see the box model by inspecting your webpage, and scrolling to the box model section. You can see that the content section - the part in the middle - is exactly 300x50 pixels. The size we set in CSS. However, the additional size is due to the extra padding and margins that were set for this object - 20 pixels on each side in our case.

This said, it is important to remember that when you set the width and height properties of an element in CSS, you are *only* setting the size of the content area. To calculate the entire size of an area, you must also include the padding and borders.

Box-Sizing Border Box

As discussed in the previous section, the total size of your element is going to be different from the width and height properties established in your CSS. The actual width of your element

follows the formula of width + padding + border = actual width, while the height follows the formula of height + padding + border = actual height.

Rather than having to calculate all of this manually each time, we have a method in CSS for establishing total box size ahead of time. This is done with the `box-sizing: border-box;` property. This is a property that is established to change the box type in CSS. Up until now, we have been using the `content-box` type, which sets the size based on the content area of our element. However, if we set our `box-sizing` property to `border-box`, it will keep all elements constrained to our established height and width without adding additional space when accounting for padding and borders.

Universal Selector

Thus far we have been selecting objects in CSS by either defining their element type, ID, or class. Another important selector to know is the universal selector, which modifies all elements in the document. This is denoted in CSS by an asterisk `*`. This is a very powerful tool, and should be used wisely.

An example of when this may be useful is if you have multiple elements in your document, but you note that they have default padding that pushes them too far away from each other. You can set this default padding to another number, or even remove it entirely, by using the universal selector.

```
*{
    margin: 0;
    padding: 0;
}
```

Due to the rules of specificity, any margin and padding set specifically in another element or class will still be active, but the default values set by the browser will be removed. This is good practice when starting a new HTML/CSS project in order to dial in your own settings as you build your webpage.

Block-Level vs. Inline-Level Elements

All HTML elements are classified in two categories: Block-Level elements, and Inline-Level elements.

A block element will typically occupy the entire available width, and it always begins on a new line. Headings are an example of this: if you place multiple h2 elements in the body of your document, each will be placed underneath the last. Giving a background color to these headings will demonstrate that their width is equal that of the entire available width of the web page. Even if you apply a different width with CSS, they will remain one below the other.

Inline elements do not always start on a new line. Instead they occupy only the space they need. An example of an inline element is an anchor tag. Adding multiple to the page will show that they appear side-by-side when possible.

Understanding the difference between block elements and inline elements will be important as we continue learning CSS.

Display Property

We can change the state of an element from block to inline or vice versa using the `display` property. One useful reason why we might do this is because inline elements such as anchors or spans do not allow for vertical margins. Even if you modify the `margin-top` property in CSS, nothing will happen due to the nature of inline elements. Block elements, however, can have vertical margins. Therefore, if you want to add vertical margins to an inline element, you can change its display type to that of a block element.

Modifying this in CSS is very easy. You can change an inline span into a block element like so:

```
span{
    display: block;
}
```

This also works the other way around, wherein you can change a block element to an inline much the same way.

```
div{
    display: inline;
}
```

Despite their inherent types, modifying the `display` property will cause each respective element to behave like their newly defined type.

There is a third display type called `inline-block`, which causes the element to spatially behave as an inline element (only taking up the space it needs), but also respect margins, height, width, and padding.

Rounded Corners

Rounded Corners on CSS elements can give a modern feel to your webpage. They are incredibly versatile and customizable. Rounded corners can be achieved by modifying the `border-radius` property.

Let's say we have a simple blue square with the `box` class. It's got some basic styling already. If we want to add rounded corners, we can do so by changing the value of `border-radius`

```
.box{
    width: 200px;
    height: 200px;
    margin: 30px;
    background-color: blue;
    border-radius: 20px;
}
```

As with any size-based parameter, the unit of measurement must be specified, and percentages are also able to be used (up to a maximum of 50%). In the case of border radius, the larger the number, the more rounded the corners will be. When the border radius equals half of the element's width and height, the element will be transformed into a circle.

You can also set individual corner radiuses by utilizing `border-top-left-radius` , `border-top-right-radius` , `border-bottom-left-radius` , and `border-bottom-right-radius` . Of course there is also shorthand for this, allowing you to specify four different values within `border-radius` for each corner. The order of input when doing this begins at top left, and works clockwise to top right, bottom right, and bottom left. For example:

```
.box{
    border-radius: 50px 10px 50px 10px;
}
```

Text Styling Properties

CSS has many properties for changing the appearance and formatting of text. Three of the most commonly used include `text-align` , `text-decoration` , and `text-transform` . Let's look at these one by one.

Text Align

Text align is used to modify the alignment of your text relative to the parent object. By default, text is aligned left. However, you can set the value of `text-align` to `left` , `center` , `right` , or `justify` in order to align the text in the chosen direction. The first three are self-explanatory. Justify will stretch the spacing of the text to take up the width of an entire line.

Text Decoration

Text decoration is a property that allows you to modify your text with various lines. For example, underline, overline, and line-through. Each of these are self-explanatory. You can also change the appearance of the line that is affecting the text. The additional parameters are color, style,

and thickness (in no particular order). So if I wanted a thin white dotted line through my text, I might style it something like this:

```
.text{
    text-decoration: line-through white dotted 5px;
}
```

Text decoration can also be used to remove certain default properties from HTML/CSS elements. For example, links by default include an underline. You can remove this by setting the `line-decoration` property of links to `none`.

Text Transform
Text transform specifies how to capitalize an element's text. Regardless of the case of the letters in the HTML you are working with, you can set the `text-transform` property to override it. `text-transform: uppercase` will make the entire text uppercase. `lowercase` will set it to lowercase. `capitalize` will capitalize the first letter of each word. This is primarily useful when the text you are working with is coming directly from a database/API and you have no control over what the raw text will look like.

Font Settings

There are three important font properties in CSS: `font-size`, `font-family`, and `font-weight`. Each is fairly self-explanatory, but we will take a deeper look at them:

Font Size
This allows you to set the size of your font. Much like other size-based properties, you can specify your unit of measurement. The standard for fonts is `pt`, or Points. The default font size of a paragraph is 12pt. Other useful measurements include `em` and `rem`. These are used for relative scaling in order to account for different screen sizes. `em` indicates 100% the size of the object's parent.

For example, if we have text inside of a div, where the div font size is set to 10pt, then setting the child text's font size to 2em will make this text 20 points in size. `em` is more useful for margins and padding, but it is still worth knowing. `rem`, on the other hand, is more useful for relative font size. while `em` is set relative to the parent, `rem` is set relative to the root of your HTML document, which is almost always the HTML body.

If we set font sizes in REM, it will allow easy consistency and parity among the text on our page, wherein we can set paragraph font size to 1rem, then headings of various sizes to 2rem, 3rem, etc. allowing our text to scale proportionally across the webpage.

Font Family
The font family allows you to select the typeface used in your font. If you want to use a font that

has two words in it, such as 'Segoe Ui', you will need to specify this inside of single quotes:

```
p{
    font-family: 'Segoe Ui';
}
```

You can also assign a backup font if the chosen font is unavailable or unsupported. Do this by simply typing the name of another font immediately after the first, separated by a comma. You can assign multiple backup fonts. The last should always be `sans-serif`.

You can assign other typefaces that are not available by default by importing it into your HTML file. The easiest way to do this is by finding a font in Google Fonts (or another site that has an embed option) and copying the embed link into the `head` section of your HTML file. From here you will be able to call the font name as normal.

Font Weight
The font weight property allows you to choose between `normal` or `bold` for your font. This will override any bolding done in HTML, meaning you can un-bold a header or strong type text, or bold text that is otherwise normal. You can also assign a number in this field between 100-900. 100 is a very thin font, while 900 is as bold as it can get. The default is 400. Note that this is dependent on the boldness options available in your chosen typeface.

You can also set this value to `bolder` or `lighter`, which will add or subtract 300 from the parent's default relative font weight. Once again, the default is usually 400, but some fonts may come with different relative font weights.

Center Elements

"How to center a div" is classically one of the most searched programming-related questions on google. Let's go over how to horizontally center an element relative to its parent container.

Let's say we have a light blue div with a height of 50%, and a width of 300px. To center it on the screen, we use the margin property. Recall that you can set vertical and horizontal margins by establishing two numbers. Since we don't need to do anything with the vertical margins, we'll set the first value in our margin property to 0. The second value will be set to `auto`, which will automatically set the horizontal margins so that the div is always centered relative to its parent element.

It is important to remember the following as a method of centering any block-level element horizontally relative to its parent:
`margin: 0 auto;`

We can also center an element vertically using a property called `align-content`. This is useful for centering text in a container and similar use cases. Important to note here that this is done on the parent-level, and will vertically align all of its child objects. To do this, go to your parent object for the elements you want to align, and add the following property:
`align-content: center;`

You can combine both of these methods to align elements to the middle of your web page and elements.

Day 3

Intermediate CSS

Background Property (Images)

There are several different properties to set and modify a background image. The ones we will be going over here are `background-image`, `background-repeat`, and `background` size`.

Images should be stored in their own directory in the same way that was covered in the HTML section. For the purposes of this section, we will assume images are stored one subfolder past the root of the project, or `~/images`. If you want to add the image to your CSS body rather than as an HTML element, you can do so by modifying the `body` selector in you CSS code:

```
body{
    backround-image: url("images/image.jpg");
}
```

Note the syntax - you must include `url()` even if you aren't strictly providing a universal resource link. If you are storing your image locally as discussed above, you direct your code to the relative path of the image.

If you do this now, you may note that the image repeats itself in a tiling effect. This is because the image is attempting to fill the entire background of your web page while also maintaining its aspect ratio. To resolve this, you can modify another property: `background-repeat`.

```
body{
    backround-image: url("images/image.jpg");
    background-repeat: no-repeat;
}
```

As you might guess, this removes any duplicates of the image, and only keeps the original on your page. However, a new problem arises: depending on the aspect ratio of our image, it's

most likely not going to cover our background now. We can once again modify a new property here called `background-size`, to change the way our image behaves.

```
body{
    backround-image: url("images/image.jpg");
    background-repeat: no-repeat;
    background-size: contain;
}
```

The `background-size` property has two most commonly used values: `cover`, and `contain`. If `contain` is selected, the entire image will fit within the parent element without cutting off, and will sacrifice size in order to maintain its aspect ratio. Setting it to `cover` will scale up the image to cover the entire webpage, at the cost of cropping the image in order to maintain aspect ratio.

If you have a webpage that is large enough to require scrolling, our current settings will keep the image pinned to the top, and will no longer be visible once you scroll past a certain point. This can be changed with the `background-attachment` property.

```
body{
    backround-image: url("images/image.jpg");
    background-repeat: no-repeat;
    background-size: contain;
    background-attachment: fixed;
}
```

Setting this property to `fixed` will keep the image fixed in place even if you scroll. By default, this value is set to `scroll`.

Transparent Colors, Alpha Values, and Opacity

For the sake of this section, let's create a basic HTML file that contains two divs with some basic styling. One will have the class `alpha` while the other will have the class `opacity`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Alpha and Opacity</title>
    <style>
        div{
            width: 300px;
            height: 300px;
```

```
            display: inline-block;
            margin-right: 50px;
            border: 5px solid black;
        }
    </style>
</head>
<body>
    <div class="alpha"></div>
    <div class="opacity"></div>
</body>
</html>
```

If we want to work with opacity, we will need to use a different color style. Previously we've used `rgb`, but now we will use `rgba`, which stands for "red, green, blue, alpha" - where "alpha" refers to opacity.

As before, the first three values in `rgba` control the red, green, and blue hues respectively. They can range from 0-255 inclusive. The fourth value, `alpha` is a value that can either be 0 or 1, where 1 is fully opaque, and 0 is fully transparent. The alpha value can contain decimal numbers to create slightly transparent objects.

For example, let's say we want to make a slightly transparent blue square out of our `alpha` div.

```
.alpha{
    background-color: rgba(0, 0, 200, 0.7);
}
```

In this case, you can see that our red and green values are 0, blue has some value, and alpha is 0.7, making it 70% opaque.

`transparent` is also a named color. Anything set with this color will disappear entirely. You can set, for example, `background-color: transparent;` on an object to render it invisible in your web page. This will become more relevant later.

Alternatively, there is the `opacity` property. Rather than setting an alpha value in a color, this value will set the transparency of the entire object. In the example with `rgba`, we saw that while the background color was changing in opacity, the border color remained constant. If we instead modify the `opacity` property, the entire element will change opacity, and not just the color.

Once again, `opacity` is a value between 0 and 1, with 0 being completely transparent and 1 being completely opaque. Decimal numbers are used to define percentages of transparency. We can create a purple square with our `opacity` div, and see the difference in how `opacity` functions vs `alpha`.

```
.opacity{
    background-color: rgb(128, 0, 128);
    opacity: 0.7;
}
```

Once again we have set this to 70% opacity for direct comparison. As you can see, the border's opacity is affected as well as the color. `opacity` affects the entire element, while `alpha` only affects the color.

Linear and Radial Gradients

A gradient is a combination of two or more colors, where one color will gradually morph into another over a set distance. A linear gradient transforms color in a straight line, while a radial gradient does so in a circular pattern from the center out. There is a third type, conic, which creates the gradient pattern in a cone shape. However, we will not be discussing this type in this lesson.

To begin, let's once again create an HTML file with two basic divs, each with their own unique class. I am naming the classes `linear` and `radial`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Gradients</title>
    <style>
        div{
            margin: 20px;
            width: 200px;
            height: 200px;
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <div class="linear"></div>
    <div class="radial"></div>
</body>
</html>
```

We will begin with linear gradients.

In CSS, gradients are considered to be images. So in order to begin creating a gradient, we need to add the `background-image` property to our `.linear` class, and set it to `linear-gradient()`. Within the parentheses of this property is where we create the actual gradient. In its most basic form, we can simply set two colors to fade between.

```
.linear{
    background-image: linear-gradient(red, yellow);
}
```

Note that these color declarations are separated by a comma. This will create a linear gradient from top to bottom, with our first color being on top, and our second being on the bottom. More colors can be added in this way. The order will always align the colors from top to bottom in the order they are input into the gradient.

To change the direction of the gradient, you can specify a direction *first* before the colors. By default, the gradient direction is `to bottom`, meaning colors start at the top and go toward the bottom. If we change our gradient direction to `to right`, we will notice now that red is on the left, and yellow is on the right.

```
.linear{
    background-image: linear-gradient(to right, red, yellow);
}
```

Other directions for gradients include `to top`, `to left`, or if you want to create a diagonal gradient, `to bottom right`, `to top left`, etc. If you want to be very specific with your gradient direction, you can specify a radial angle, such as `50deg` (for 50 degrees). `0deg` is the same as `to top`, and the degrees move clockwise as the number gets larger until the maximum of 360.

You can also set percentages of the colors in your gradient. For example, let's add a third color, black, to our gradient, and set its percentage to 30%:

```
.linear{
    background-image: linear-gradient(black 30%, red, yellow);
}
```

When you assign a percentage to a color, that color will take up the defined percentage of the object before beginning its transition to the next color. In this case, the black color in our gradient takes up 30% of the div before beginning to fade to red, thus squashing red and yellow further down.

Percentages work by defining the start of the gradient as 0%, and the end of the gradient at 100%. The direction of this depends on the direction of the gradient, but for this example we will say that we have the default gradient direction of `to bottom`. If we set percentages for red and yellow, we can get a better feel for this:

```
.linear{
    background-image: linear-gradient(black 30%, red 50%, yellow 75%);
}
```

Now as we have established, our black color will begin at 0% and work its way down to 30% before beginning its transition. Between 30% and 50% will be our transition to red, and then from 50% to 75% will be the transition to yellow, with yellow taking up the final 25% of the shape.

We can also use percentages as full color stops. Let's say we were to add another parameter for `red` into the mix:

```
.linear{
    background-image: linear-gradient(black 30%, red 50%, red 75%, yellow
75%);
}
```

If you're playing along, you'll notice that we now have a hard transition from red to yellow at the 75% mark of our div. This is because the red gradient is assigned to go specifically from 50% to 75%, and then yellow begins at 75%, leaving no space for a smooth transition. We can do the same thing by adding another black stop at 50%, and now we have three solid colors with no transition between them. This can be useful as an advanced coloring technique.

Radial gradients work in a very similar way, where we define `radial-gradient` instead of linear. In the case of radial gradients, your first color will be in the center, and each additional color will be added moving outwards.

```
.radial{
    background-image: radial-gradient(red, yellow);
}
```

Instead of defining directions as with linear gradients, we can define where the radius of the gradient begins. For example, we can set `at top`, `at bottom`, `at top left`, etc.

Radial gradients also support percentages, where 0% is the radial center of the gradient, and it moves toward 100% as you get closer to the edges of the element.

Gradients can also be stacked onto one element. You can add a linear and a radial gradient in the same space, allowing for more complex colors. Syntax for adding multiple gradients looks like this:

```
.radial{
    background-image: radial-gradient(red, yellow), linear-gradient(green, brown);
}
```

If you tried applying this now, you will notice that the linear gradient cannot be seen yet. This is because the radial gradient is currently occupying the entire space of the div. To change this, we can utilize the `transparent` color that we learned about in the previous section. We can change the radial gradient's second color from yellow to transparent, and now notice that the linear gradient pops through behind the red of the radial gradient.

Since gradients are considered images, you can also use this same technique to combine images with gradients to create some wild effects. Recall that images can be inserted by using the `url()` function.

Shadows

Shadows are crucial in modern UX design, because they give your elements a sense of depth, making them more engaging and visually appealing. For the sake of this lesson, we'll establish a basic HTML file with a simple div for demonstration:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Shadow</title>
    <style>
        div{
            margin: 50px;
            width: 300px;
            height: 300px;
            background-color: aqua;
        }
    </style>
</head>
<body>
    <div></div>
```

```
</body>
</html>
```

We can add shadows to elements with the `box-shadow` property. There are several parameters to be tweaked with `box-shadow`, so we will go over the most useful ones.

The first two parameters for `box-shadow` are mandatory: X offset and Y offset. Let's say we establish the following:

```
div{
    box-shadow: 20px 10px;
}
```

This will offset our shadow 20 pixels to the right, and 10 pixels down from our element. Important to note here that the coordinate system in CSS establishes 0,0 at the top left corner of the element. Meaning when we set our box shadow to `20px 10px`, it is moving 20 pixels to the right from the top left corner, and 10 pixels down. Adding negative values will make the shadow move to the left and up respectively.

After defining the offset of the shadow, we can define additional parameters. For example, we can set the color to grey. This is done by adding the color after the offset, no comma needed.

```
div{
    box-shadow: 20px 10px grey;
}
```

Between the offset and the color, we can establish a third measurement parameter. This parameter is for feathering, which makes the shadow blur outward. The higher the value of this parameter, the more blurry it becomes - thus appearing more like a shadow and less like a solid shape.

```
div{
    box-shadow: 20px 10px 15px grey;
}
```

Now this should look more like a 3 dimensional shape casting a shadow, and less like two boxes.

After this feather, we can define the spread of our shadow. This will change its size depending on if a positive or negative number is entered here. The following will make our shadow slightly larger:

```
div{
    box-shadow: 20px 10px 15px 5px grey;
}
```

Finally, we can inset the shadow by adding the `inset` keyword after our color. This makes it appear as though the shape is recessed into the site rather than floating above it.

```
div{
    box-shadow: 20px 10px 15px 5px grey inset;
}
```

You can have multiple shadows on the same object to create more complexity and depth. This can be done by separating the two sets of parameters by a comma.

```
div{
    box-shadow: 20px 10px 15px 5px grey inset, 5px 5px 3px grey;
}
```

It is highly encouraged to play around with this property and its parameters to see what everything does.

Combinators

Combinators are a method of selecting elements based on their relationship to other elements. HTML elements are often nested within other elements, such as a heading and paragraph being nested inside of a div. This creates a hierarchy similar to a family tree, where elements containing other elements are called parents, and elements nested within elements are called children. It is possible for certain elements to be both parent and child at once, as in the below example:

```
<html>
    <body>
        <div>
            <h2>This is a heading</h2>
            <p>This is a paragraph</p>
        </div>
    </body>
</html>
```

In this case, the header and paragraphs are children of the div, which is itself a child of the body, which is a child of the html element. We would also refer to the h2 and paragraph as

siblings, since they are both nested with equal hierarchy as children of another element (the div).

Combinators allow for selecting objects and elements based on this relationship. Specifically in this section, we will discuss `descendant` , `child selector >` , `next-sibling +` , and `subsequent-sibling ~` .

Let's take a look at the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Combinators</title>
    <style>
        header{
            background-color: beige;
            padding: 20px;
        }
        div{
            background-color: bisque;
            padding: 20px;
        }
    </style>
</head>
<body>
    <header>
        <h2>Combinators</h2>
        <p>Lorem Text</p>
    </header>
    <div>
        <h2>Another Heading</h2>
        <p>Lorem Text</p>
    </div>
</body>
</html>
```

Now let's say we want to modify the h2 specifically within the `<header>` element. How can we go about just selecting it and not the other h2 in the div? We could assign it a class or ID, but assigning multiple classes is clunky and bad practice. We can use a combinator to isolate just the h2 that is a child of the header by specifying the following:

```
header > h2{
    color: blue;
}
```

This syntax of `parent > child{}` allows us to select specific children of the specified parent object(s). If the header had multiple h2 headings, all of them would be selected, but *only* the ones that are children of the header.

Let's say we have multiple headers, each with their own classes, and we only want to select the h2 element that is a child of the header with the "container" class. We would do this with the following syntax:
`header.container > h2{}`

So to select only the child of a specific object, where the specific object has a class, the syntax is `parent.class > object{}`

It is important to note that this method only affects direct children. For example, let's say we have the following setup:

```
<div>
    <h2>A Heading</h2>
    <article>
        <p>Lorem Text</p>
    </article>
</div>
```

How would we go about selecting the paragraph that is a child of the article contained in the div? We would select `article > p{}`. However, we can also combine this selection to create a more specific selector, such as `div > article > p{}`. Combining selectors in this way increases the specificity. Meaning the more combined selectors there are, the more precedence the styling of this selection will take. Continuing on the example above, let's say our CSS looks like this:

```
article > p{
    color: green;
}
div > article > p{
    color: aqua;
}
```

The color of our paragraph will be set to aqua and ignore green, even though both selectors point to the same element. This is because the second selector containing `aqua` contains more

specific instructions, and is therefore higher in specificity.

Similar to the child selector is the descendant selector. The difference is that this combinator selects all descendants of the specified element, and not a single one as with child selectors. Syntax looks like this:

```
div h2{}
```

Here, instead of using the `>` to specify a single element, we instead simply use a space to define "select all H2 elements inside of a div". Currently this is identical to the child selector, wherein we can use `div > h2{}` to select the same. However, descendant selectors also select objects that are nested within other elements. As with the example above with the paragraph nested in an article in the div, we can select this paragraph with `div p{}` . The fact that it is nested in the article is irrelevant, because it is still a descendant of the div.

Let's look at another example HTML document:

```
<body>
    <header>
        <h2>This is a heading</h2>
        <p>This is a paragraph</p>
        <p>This is a paragraph</p>
        <p>This is a paragraph</p>
        <p>This is a paragraph</p>
    </header>
</body>
```

As we can see here, the h2 has the same parent as the paragraphs, making them siblings. We can use the `subsequent sibling` combinator to select all of the paragraphs.

```
h2 ~ p{}
```

In this case, this will select all paragraph elements that are subsequent siblings of an h2 element - meaning it will only affect paragraphs that come *after* the h2. So if we had a setup like this:

```
<p>This is a paragraph</p>
<h2>This is a heading</h2>
<p>This is a paragraph</p>
```

then selecting a paragraph using the subsequent siblings combinator will only affect the *second* paragraph, as it comes after the heading.

There is also the `next sibling` combinator, which only selects the sibling that comes immediately after the specified element. For example, if we only wanted to select the single paragraph immediately following the h2 in the above example, we would specify `h2 + p{}` .

All of these combinators can be combined together to create some very specific selections. It is encouraged to try them for yourself to see what can be done using combinators.

Attribute Selectors

Just as with classes, IDs, and combinators, it is also possible to select HTML elements by their attributes. This means we can select links with specific targets, or images with specific alt text. However, attribute selectors are typically used to style input elements.

Let's take a look at the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Attribute Selector</title>
    <style>
        form{
            padding: 20px;
        }
        input{
            margin: 5px;
        }
    </style>
</head>
<body>
    <form>
        <input type="email" placeholder="Enter your email"><br/>
        <input type="password" placeholder="Enter your password"><br/>
        <input type="submit" value="SIGN IN">
    </form>
</body>
</html>
```

Let's say we want to style the email and password fields differently from the sign in button. We wouldn't be able to use the `input` selector, or even any currently known combinators, as all

input elements would be selected this way. Instead, we can use the attribute selector to select these elements separately.

Attribute selector syntax utilizes square brackets, as such:

```
input[type="email"]{
}
```

Here you can select any given attribute by first specifying what type of element it is, and then targeting the specific attribute of that element in the square brackets.

Now in this case, we *could* call it good there and write our styling, then select the password field and copy/paste the styling over to it. However, if you want to select multiple attributes to be styled the same way, all you need to do is separate them with a comma.

```
input[type="email"], input[type="password"]{
    height: 25px;
    width: 150px;
    border-radius: 30px;
    border: 1px solid black;
    padding-left: 10px;
}
```

This cleans up your code and makes it significantly more readable.

While attribute selectors are typically used for forms and inputs, they can theoretically be used for any HTML element with an attribute.

Pseudo-Classes

Pseudo-classes are a method of styling elements based on their specific states, such as if your mouse is hovering over it, or if a checkbox is checked. Pseudo-classes are an essential part of a responsive and user-friendly UI.

Let's look at some CSS styling for an anchor:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Pseudo-Classes</title>
    <style>
        a{
```

```
            margin: 10px;
            width: 250px;
            text-align: center;
            display: inline-block;
            text-decoration: none;
            color: white;
            text-transform: uppercase;
            background-color: tomato;
            padding: 1em;
            border-radius: 10px;
            font-weight: 600;
        }
    </style>
</head>
<body>
    <a href="#">This is a link</a>
</body>
</html>
```

Right now this button looks nice enough, but if you hover over it or click on it, nothing seems to respond in the UI. We can begin to change this by modifying one of the most important pseudo-classes: hover.

Pseud-classes are selected by first specifying the element that you would like to select, and then specifying the pseudo-class you want to modify. In this case, we want to change the link's appearance when the mouse hovers over it. We can select the hover pseudo-class by adding the following OUTSIDE of our current `a` styling:

```
a:hover{}
```

So syntax for pseudo-classes is:

```
selector:pseudoclass{}
```

Any styles that you apply within the brackets here will only apply when the user hovers their mouse over the link. It is important to note that you don't need to re-style this element from scratch - you only need to specify the properties that will change when the conditions for the pseudo-class have been met.

Another important pseudo-class especially for links is `:visited`. With this pseudo-class, the styling will change if the user has clicked the link at least once. For example, we could change the link to actually redirect to a website, and then set the following styling:

```
a:visited{
    background-color:blueviolet;
```

```
}
```

Now you can test for yourself that when you click the link and return, the color for the button is forever changed.

Similarly, there is the `:active` pseudo-class, which changes the styling as the link is actively being clicked.

Here are a few more useful pseudo-classes and a general description of what they are used for:

- `:focus` - This is used primarily for input fields. When the input field is selected and ready to be typed in, the `:focus` pseudo-class applies. This is useful for indicating where the user is typing at a glance. By default, forms in focus have an outline. You can set this to `none` by using the focus pseudo-class.
- `:invalid` - This is also used for input fields. The invalid pseudo-class will trigger when you attempt to input an invalid value into a field.
- `:valid` - This is the direct opposite to the above - styling under the valid pseudo-class will apply when the selected input field has received valid input.
- `:checked` - This is specifically for checkboxes. Styling will only apply if the checkbox is actively checked.
- `:first-child` - This is particularly useful for lists, but can be used with any element that has multiple children. Styling here will only apply to the first child of the selected element.
- `:last-child` - Exactly the same as above, only it affects the very last child of the selected element rather than the very first.
- `:nth-child(#)` - This functions the same as the above two, only you can select which specific child in the list of siblings should be selected. The `#` in parentheses should be replaced with the number child you want selected. Alternatively, instead of a number, you can pass in certain keywords such as `even` or `odd` to select every even- or odd-numbered child of the selected element. Finally, you also have the option to pass in `3n` to select every third element. Use whatever number you'd like before the `n` to select every `n`th element.

You can find more about pseudo-classes in the documentation.

Transitions

This section will cover how to create a smooth transition for hover effects. If you've been following along, then you noticed in the last lesson that when we styled the link, the transition between hover and standard was abrupt and a bit jarring. Transitions are the way to keep everything nice to look at with respects to hover effects.

Let's say we're working with the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Transition</title>
    <style>
        div{
            height: 300px;
            width: 300px;
            background-color: brown;
        }
        div:hover{
            border-radius: 40px;
            background-color: chocolate;
        }
    </style>
</head>
<body>
    <div></div>
</body>
</html>
```

Here we have a div with a hover effect that changes its color and rounds its corners. We can give this effect a transition animation by adding the `transition` property to the hover pseudo-class. Transitions are measured in time, which we haven't worked with thus far. You can set the transition to last one full second with `transition: 1s;` , or you can specify an amount of milliseconds with `transition: #ms;` .

You may notice that when you remove your mouse from the div, it immediately jumps back to its pre-hover state with no transition. To ensure it transitions evenly both ways, it's best practice to place the transition property in the default, non-hover selector rather than the hover pseudo-class.

You can also specify which specific properties in the element should follow the rules in this transition. For example, if we only wanted the border radius to transition after 1 second, but everything else to be sudden we could define the following:

```
div{
    transition: 1s border-radius;
}
```

Additionally, you can set different transition times for different properties by separating them with a comma.

```
div{
    transition: 1s border-radius, 3s background-color;
}
```

You can also set a speed curve for your transition. This is similar to keyframe easing curves in any animation software. The default is `ease`, which provides a smooth transition between the two states. However, you can set this to a few other states, such as `linear`, which maintains a consistent speed from start to stop. `ease-in` will start the transition slowly, and then end linearly once it reaches max speed. `ease-out` does the same, but in reverse. `ease-in-out` slows down at the start and end of the transition, and is preferred by most developers.

Positions

Thus far we have been primarily working with elements in a static position, meaning they will stay exactly where they are on screen. However, elements have five different positions, each with a unique purpose. Each position other than static has five directional properties that can be altered: top, bottom, left, right, and z-index. Let's briefly discuss the other four positions:

Relative
Relative positioning allows you to position the object relative to where it *would* be if it were a static object. Changing the `top` and `left` properties by any given amount will push the object to down or to the right relative to where it would be positioned if it were static. You can also change the other parameters, but this will usually push the element off screen. It is difficult to relay how this works via writing. It is encouraged to play around with this setting to see how it works for yourself.

Fixed
Fixed position means the element will stay in place and on the top layer even if the page is scrolled. This is useful for things like pop-ups. In this position, the directional properties are no longer relative to the element, but to the entire page. For example, setting `right: 0;` in the fixed position will set the element's right side 0 pixels away from the right edge of the screen - in other words, it will be right up against the right edge of the page.

You can set the left and right positions to 50% each, but you will notice that the element itself is not centered. This is because, as discussed previously, coordinates are based on the top right corner of the element, and not its center. As such, the top right corner will be perfectly centered, but the element will not be perfectly centered.

To rectify this, we need to set the `translate` property. `translate` effectively modifies the element's position relative to its own size. If we set `translate: -50% -50%;`, then it will push

the element back and up by 50% of its own size, allowing for a perfectly centered element.

Sticky
A sticky element behaves similarly to a static element, with the notable exception that if you scroll past it, it will stick to the specified sticking point and follow you down the page. If you scroll back up, it will return to its regular position. You can set the sticking point by specifying the `top` property.

Absolute
This is by far the most important position. An element with `position: absolute;` will always be positioned relative to its nearest ancestor that has a specified position. For example, take this HTML body structure:

```
<body>
    <div class="parent">
        <h2>Parent Element</h2>
        <p>Lorem Text</p>

            <div class="child">
                <h2>Child Element</h2>
            </div>
    </div>
</body>
```

We have a parent div with its own elements, and it contains a child div that has an element as well. In our CSS, the child element has a position value of `absolute`. In order to have an absolute value for an element, its parent needs to have a defined position as well. In this case, we will use `relative`.

Now if we change any of the directional properties in the child element, it will position itself according to the location of the parent element. This means that if we set `top: 0;` in the child element, it will align its top with the top of the parent element, NOT the page. The same goes for all other directions. The position will be relative to the parent, and the rest of the page will be effectively ignored by the child.

If the parent is not set to `relative` positioning, then the child element will position itself relative to the webpage's body, not its direct parent. Much like an object with a sticky position, `absolute` objects will stay on the top layer unless otherwise defined.

You can change the layers that an object is on (for any position) by changing the `z-index` value. By default, everything is set to `z-index: 0;`. However, if you set a value to -1, it will go behind the other layers. Similarly, setting the value high will ensure no other objects obstruct it. You can manually set `z-index` layers so your site is structured exactly as you need it to be.

Pseudo-Elements

A pseudo element allows you to target specific parts of elements in CSS. Where pseudo classes allow for modifying states, pseudo elements modify content. Pseudo element syntax involves using two colons instead of the one we used for pseudo classes.

```
div::before{}
```

One particularly useful pseudo element is `::first-letter`, which only modifies the fist letter of the selected element. This can be used to create a drop cap or otherwise stylized letter.

```
p::first-letter{
    font-weight: bold;
    font-size: 30;
}
```

Here is a list of some other useful pseudo elements and what they're for:

- `::first-line` - as with first letter, this modifies the first line of text in the selected element.
- `::placeholder` - this specifically adds style to the placeholder of an input field.
- `::before` and `::after` - These are arguably the most important pseudo elements. These essentially add styling immediately before or after any given element. This essentially creates an empty element in the specified position, and allows you to stylize it as you see fit. This is useful for a multitude of things, including custom bullet points, extra flair on your site, animations, etc.
  With these pseudo elements, you must define everything from scratch - there is nothing inherently included in them. The first property to be added is `content:` which determines what will actually go in this space. This is always a string value, so if you don't want any actual text to go in your pseudo element, then define content as an empty string with `content: '';`. This also means that it has no inherent size. You'll need to set its position to `absolute` and its parent position to `relative`, then manually set the size. From here, you can stylize it as you would anything else in CSS.

Variables

As with any good programming language, CSS has the ability to define variables. These are also known as custom properties. A good use case for CSS variables would be if you have a specific RGB/HEX color value to be applied in multiple places. Instead of copy/pasting the value each time, you can set value as a custom property, and use it in place of the value in your styling.

To create a variable in CSS, you must define it in `:root`, which is the element at the very top of your style document - even higher than universal `*` or `html`. To define a variable, you precede it with two dashes, followed by the variable name.

```
:root{
    --primary-color: #828aff;
}
```

To use this variable in your CSS stylings, you must call it with the syntax of `var(--variable-name)`. For example, now that we've defined the color above, let's use it:

```
button{
    background color: var(--primary-color);
}
```

Day 4

Project Notes

The beginning of day 4 involved creating a simple multi-page website. It was mostly an application of skills learned thus far, but there were a few extra tidbits that had not been taught previously that I felt were worth noting here:

- `scale` - This is useful especially for hover effects. It takes one argument, which is a number to scale up (or down if you choose a negative number) by. For example, if you set `scale: 1.1;` on a `:hover` element, then your element will scale up by 10% when hovered. This is useful for scaling X and Y axes simultaneously and ensuring they remain relative.
- `calc()` - This parameter is useful for setting sizes that may be arbitrary or would require a bit of math to get right. It takes in multiple sizes (even at different measurement units) and calculates them accordingly. You can use standard arithmetic operators for this function. The example used in the project was to keep content below the nav bar, we had to calculate the size of the nav bar by factoring in its padding, font size, and border. So the padding was set to `margin-top: calc(50px + 1rem +2px)`
- `min()` - This parameter is useful for responsive web design across multiple devices and screen sizes. It is a size-based parameter, and can be used in place of an actual number, such as `1000px`. `min()` takes in two sizes, and will always use the smaller of the two depending on screen size. An example used in the project was to add a video player to the site. The width was set to `width: min(1100px, 100%);` - thus setting the minimum size to either 1100px if possible, otherwise 100% of the screen width.

- `aspect-ratio` - This allows you to set the aspect ratio of your selected element. In the example above with the video player, we set the width to our `min` value, and instead of manually setting a height, we set the aspect-ratio property in order to keep everything to scale. In this example, it was `aspect-ratio: 16 / 9;`

Animations

There was only one lesson in this section, but I had to keep the headers consistent across my notes. Hopefully you will forgive me. For the sake of this section, we will assume we already have a simple stylized div element in place. Feel free to create your own.

Animations in CSS are used when a transition isn't enough for your visual motion goals. I.E., if your animation requires multiple steps rather than a start and end point, or if you need something in constant motion such as decorative elements.

Animations in CSS are first created by defining them with the `@keyframes` rule. This will be followed by the name of your animation. It can be called anything you'd like, but standard naming conventions apply. We'll call ours `spin`.

Within this keyframes rule, we need to define each step in the animation. Steps are expressed as percentage values. 0% is the beginning of the animation, and 100% is the end. Below is boilerplate setup for a basic animation.

```
@keyframes spin{
    0%{
        /*animation instructions will go here*/
    }
    100%{
        /*animation instructions will go here*/
    }
}
```

Since we are spinning this element, this logically means that our element will be spun around completely by the time the animation reaches the end. Therefore, in the instructions for the 100% mark, we will modify the `transform` property. We'll also update the border radius in order to turn it into a circle.

```
@keyframes spin{
    0%{

    }
    100%{
        transform: rotate(360deg);
```

```
        border-radius: 50%;
    }
}
```

Now we can assign this animation to our div by adding `animation-duration` and `animation-name` to its styling:

```
div{
    /*There will be some other styling here*/
    animation-duration: 3s;
    animation-name: spin;
}
```

Now when we load the page, the animation will play exactly once, and then the element will revert back to its original state.

We can increase the complexity of the animation by adding additional instructions between 0 and 100%:

```
@keyframes spin{
    0%{

    }
    50%{
        scale: 2;
    }
    100%{
        transform: rotate(360deg);
        border-radius: 50%;
    }
}
```

Recall that the `scale` property modifies the element's X and Y coordinates proportionally, and in multiples relative to the element's size. Thus in the above code, the element will double in size at exactly halfway into the animation, and then shrink back down to 1x its original size by the time it gets to 100%. If you want it to maintain its size all the way through to the end of the animation, you will also need to add `scale: 2;` to the 100% instructions. This is similar to adding keyframes in any animation software.

Similarly, we can add `transform: rotate(0)` to our 50% instructions to keep the element from rotating until it reaches its full size in the animation. As long as you can understand that percentages act as keyframes, and properties therein need to be maintained in a similar

manner, animation should be fairly simple to understand. Any CSS property can be modified within an animation.

Let's go over a few other animation properties to be added to your element's styling (not the animation instructions):

- `animation-timing-function` - This allows us to define our speed curves for each step of the animation. Options include `ease` , `linear` , `ease-in` , `ease-out` , and `ease-in-out` . These work the same as when we discussed speed curves before, only the curves will affect transitions between keyframes of your animation, not just the start and end.
- `animation-delay` - this will define how long the animation should wait before playing once the page is loaded.
- `animation-iteration-count` - this determines how many times the animation will play. The default value is 1. This is unaffected by delay - once it starts playing, it will loop instantly by the amount of times specified. You can also set this value to `infinite` to keep the animation playing as long as the page is loaded.
- `animation-direction` - this allows you to play your animation forwards or backwards. Default is `normal` , but you can specify `reverse` here to have your animation play backwards. You can also set this value to `alternate` to have your animation play both forwards and in reverse alternating on each iteration.
- `animation-fill-mode` - this determines the final state of your element after the animation has finished playing. You can set this value to `forwards` to have the element stay in its 100% animated state infinitely after the animation is done playing.
- `animation-play-state` - This allows the user to interact with your animation, causing it to play and stop on certain events. For example, in our normal element CSS, we can set `animation-play-state: running;` . But then, on `.element:hover` , we can specify `animation-play-state: paused;` thus preventing the animation from playing as long as the mouse cursor is hovering within our element, then continuing to play when the mouse cursor is no longer hovering.

All of the above values, including `animation-duration` and `animation-name` can be combined into a single line of code by using the `animation:` property. THE ORDER MATTERS HERE. Order is as follows:

```
animation: duration name timing-function delay iteration-count direction;
```

Syntax above is correct as well. No delineators between each parameter aside from a space. Just be sure to use the correct measurement indicators in place of the parameter name.

Day 5

Horizontal and Vertical Alignment

For all of day 5, we are going to be focusing on flexbox and responsive web design. We will also be utilizing external CSS henceforth. To begin with flexbox, you will assign `display: flex;` on the parent element of whatever object(s) you would like contained within the flexbox.

Let's say we have some HTML that has five simple divs with a bit of basic styling.

```
<body>
    <div class="box">1</div>
    <div class="box">2</div>
    <div class="box">3</div>
    <div class="box">4</div>
    <div class="box">5</div>
</body>
```

and their styling:

```
.box{
    height: 200px;
    width: 200px;
    background-color: #0071FF;
    font-size: 40pt;
}
```

To utilize flexbox on these divs, we will need to specify `display: flex;` on their parent element. In this case, it's `<body>`.

```
body{
    display: flex;
}
```

Immediately upon doing so, you will notice that our divs are all lined up side-by-side rather than being stacked on top of each other. While previously we have learned about `block`, `inline`, and `inline-block` display styles, we have now entered the world of `flex`, which works differently from all the previous types.

The most important concept in flex layout is the axes on which the items are positioned: the main axis, and the cross axis. The main axis runs centered horizontally across the length of the flexbox, and the cross axis runs centered vertically across the height of the flexbox.

To position elements along the main axis, you will utilize the `justify-content` property - still in the parent element of whatever content you are justifying. This property has a few options: `flex-start` - which begins element alignment at the start (or left) of your main axis. `flex-end` pushes everything to the end (or right side) of the main axis. `center` aligns everything to the

center. That said, centering an element is as easy as adding two lines of code to its parent element:

```
.parent{
    display: flex;
    justify-content: center;
}
```

Aligning vertically means aligning across the cross axis rather than the main axis. To do this, we will first need to give our parent element some height. We will set the `min-height` to 800px for this demonstration. Distributing flexbox items across the cross axis requires using the `align-items` property. This property has the same basic values as `justify-content`: `flex-start`, `flex-end`, and `center`. This said, to truly center an object vertically and horizontally, you only need three lines of code in its parent element:

```
.parent{
    display: flex;
    justify-content: center;
    align-items: center;
}
```

Flexbox, true to its name, is flexible. Therefore this centering will remain true even if the screen size changes.

Flex Direction and Gap

`flex-direction` is an important flexbox property that controls the direction of the main axis. Its default value is `row`, which makes all content aligned to this axis flow from left to right. However, `flex-direction` can be set to other values, such as `row-reverse`, which moves everything to the right, and makes everything flow from right to left.

`column` makes the main axis go from top to bottom, thus rearranging all content as such. This also causes the main axis and cross axis to switch places, meaning if we want to center content in `column` direction, we will need to use `align-items: center` instead of the usual `justify content` as described in the previous section.

The `gap` property allows you to add space between elements inside of a flexbox. This is similar to margins, except now you don't have to take into account all of the extra measurements to get things just right. Adding a gap of 20px will simply add 20px of space between each object.

Space Between, Space Around, Space Evenly

The `space-between` parameter is another option to be used for `justify-content`. Using `justify-content: space-between;` in a flexbox setting will cause all elements to fill up the entire space of your flexbox from left to right, evenly spacing all other elements in between. Note that this will *always* cause the first and last element to touch the edge of your flex box. Adding a gap will not change this.

`space-around` works similarly to the above, only the first and last elements will not touch the edge. Distance from the edge will be equal to the distance between each of the other elements. To give a visual representation, `space-evenly` causes elements to behave like this:

```
-E--E--E-
```

Note how the space is the same, but because there are elements in the middle with padding on both sides, the gap is doubled for center elements. As such, the gap on the edges will be visually halved. To get around this, you can use `space-evenly`

`space-evenly` technically adds double the gap on the left and right sides of the edgemost elements, thus giving it visually identical gaps to the center elements. To give another visual representation, `space-evenly` will look something like this:

```
--E--E--E--
```

All of these values can also be used on the `align-items` property to align your content across the cross axis in addition to the main axis as we've seen here so far.

Flex Wrap | Align Content vs Align Items

The `flex-wrap` property gives you a responsive layout that will align your items on the next line of the flexbox if you don't have enough space. This property can be set to `wrap` or `nowrap`, with the option of `wrap-reverse` in there to make your items start at the bottom of your flexbox and wrap upwards given size constraints. `nowrap` will simply resize your flexbox items with the screen size rather than wrapping the content responsively.

When there are multiple items in a flex box, or if you have content that is wrapped, you may notice that there is a large gap between the first and second lines (and third and fourth etc. depending on your flexbox size and contents) This is because as more lines of content are added to the flexbox, more main and cross axes are added as well. As such, `align-items` will affect each individual line in your flexbox, but not all items in the box as a whole.

To modify the spacing of all the content in the flexbox, we will need to use `align-content`. By default, this is set to `space-around`, which is why the gaps between lines are so prominent.

The `align-content` property can take all of the same basic values as `justify-content` and `align-items` as discussed in the previous sections.

The main takeaway here is that `align-items` will align content based on each individual axis inside of the flexbox. `align-content` aligns all content within the flexbox together. Ultimately, a perfectly centered layout will include these four lines of CSS:

```
.parent{
    flex-wrap: wrap;
    justify-content: center;
    align-items: center;
    align-content: center;
}
```

Column Gap vs Row Gap

When you have a scenario where there are vertical and horizontal gaps, you can split the `gap` property into `row-gap` and `column-gap`. This will allow you to have finer control over the vertical and horizontal spacing of your flexbox objects. You can of course still use `gap` to specify the same value for both directions.

Flex Grow & Flex Shrink

An alternative to wrapping objects responsively in a flexbox is having them shrink/grow responsively. If you don't have wrapping enabled, flexbox items will be resized automatically depending on your flexbox/screen size. You can dial in this setting by modifying the `flex-shrink` and `flex-grow` properties *on the items themselves* (not their parents)

For example, if we have items in a flexbox with `flex-shrink` set to 0, it will prevent shrinking during resizing, and instead will cause the items to overflow the container. The default value for this property is 1, thus likening it to a binary switch turning the shrinking property on and off. This can be useful to set individual shrink values to different items in a flexbox. Disabling flex shrink on an image, for example, will keep it from distorting when the flexbox gets too small.

Conversely, `flex-grow` will cause elements to stretch along the main axis, taking up as much space as allowed by gaps and margins. Once again, this is modified at the element level, and is a binary switch. Setting this to 1 will allow elements to grow along the main axis, and setting it to 0 (default value) will keep them from growing.

Both of these properties, `flex-grow` and `flex-shrink` , can be used as multipliers in addition to binary switches. Since they work on an individual element level and can be dialed in separately, you can set one value to 5 (for example) and it will shrink/grow 5x faster than any of the other elements that are otherwise set to 1. This is useful for dynamic web design.

Combine Flexbox with Min Width & Max Width

`flex-shrink` becomes even more powerful when combined with minimum & maximum sizes. Then you can define when the growth of your boxes should stop. Thus you can set values such as this:

```
.box{
    flex-grow: 1;
    max-width: 300px;
}
```

And now you have a maximum size to be set, but still a well-responsive element. The same goes for `flex-shrink` and `min-width`. Once your screen gets small enough, your elements will stop shrinking and instead will overflow your flexbox. Element overflow should be avoided when possible; thus it is possible to combine shrink/grow with flex wrap by using a media query:

```
@media(max-width: 575px){
    body{
        flex-wrap: wrap;
    }
}
```

More info on media queries in the next section.

Media Queries

Media Queries allow your website to adapt to different screen sizes and devices. To add a media query to your site, you'll use the `@media()` keyword. Within the parentheses you can use properties like `max-width` and `min-width` to apply CSS rules based on the screen width. This allows us to create break points to change how the website responds based on the screen's size.

A way to demonstrate this would be with the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Media Query</title>
    <style>
        body{
            background-color: grey;
```

```
        }
        @media(max-width: 1000px){
            body{
                background-color: blue;
            }
        }
    </style>
</head>
<body>

</body>
</html>
```

This will create a mostly empty webpage, with the exception of the background color. At full screen, this page will display a grey background color. However, once the screen width becomes small enough (under 1000px) the background color will change to blue. You can create multiple media queries, thus creating multiple break points for different screen sizes.

Media queries are particularly useful in such cases as hiding your nav bar once the screen gets too small, or changing/resizing a graphic at certain sizes. You can even go so far as to change your entire layout style from grid to flexbox depending on the screen size.

You can also use the `min-width` property inside of a media query to cause your code to only apply when the screen is *smaller* than the specified size. This is typically referred to as "mobile-first" development, as the site is built around a certain smaller size first.

A few typical breakpoints include the following:

- `min-width: 1200px` for desktops and large laptops
- `min-width: 992px` for normal-sized laptops and desktops
- `min-width: 768px` for landscape tablets
- `min-width: 600px` for portrait tablets and large phones
- `max-width: 600px` for smartphones and smaller

Extra Tips

`align-self` is a flexbox property that is used on the *element* level rather than on the level of the parent. This can take all of the basic flexbox alignment properties, such as `flex-start`, `flex-end`, etc. When `align-self` is used, only the singular element it is used on will be affected by its alignment selection.

Unfortunately there is no similar option for `justify`, as `justify-self` is a CSS grid property (to be learned in the next day's lessons) Instead, you'll have to use the old school margin option

of `margin-right: auto;`. This is useful if you want your company logo on one side of a nav bar, and your options on the other side (as an example)

Day 6

Day 6 General Learning

Columns and Rows

Today is going to be all about the CSS grid, which is the next step up from flexbox content.

To use the CSS grid, you will once again set the display property in the parent element of the objects you wish to include in your grid. In this case, we will use the example of multiple divs in the body of our HTML file. In CSS, we will set the display of the body to `display: grid;`.

A CSS grid needs columns and rows. If you don't declare them yourself, the browser will create them automatically. In our case, we have one column, and rows equal to the amount of divs we have (nine, in my case). To establish columns, you will need to use the `grid-template-columns` property. This is not as simple as defining a single number in this property to declare how many rows you want. Instead, you must input measurements.

```
body{
    display: grid;
    grid-template-columns: 200px 200px 200px;
}
```

In this case, we are establishing three columns, each being 200px wide.

While it isn't strictly necessary to manually establish rows as well, you can do this with `grid-template-rows`. This functions similarly, where you establish measurements based on each row size.

```
body{
    display: grid;
    grid-template-columns: 200px 200px 200px;
    grid-template-rows: 200px 200px 200px;
}
```

You can once again specify the gap between your rows and columns with `gap: [#]px;`.

Alignment within a grid cell

With the grid layout, you don't need to specify hard pixel values for your rows and columns. If you want your grid to take up the entire size of the parent object, but still be evenly divided, you

can divide your rows and columns in to fractions. Where before we were establishing our rows and columns as `200px`, we can instead set each value to `1fr` - or 1 fraction. This will evenly divide our grid into space equal to the number of fractions used, and the grid itself will also take up the entire width of the parent object. Note that this only allocates more space for your objects to take up - it doesn't resize the objects themselves.

By establishing fractions rather than hard size values, our grid will automatically shrink and grow with the window size. This is similar to the `flex-grow` property in flexbox, only this resizes the grid and not the elements themselves.

You can view the exact size your grid is taking up with the developer tools. When you inspect your grid in the browser, and then click on "grid" on your parent object, each cell will be outlined in purple. This also gives you the ability to see the margins made for the `gap` property. By doing this, you will also notice that the elements in the grid are always positioned in the top left corner of each grid cell. To change the alignment of these items relative to their individual cells, you can use the `justify-self` and `align-self` properties *on the element level*. These properties use the same alignment values as the flexbox alignment properties. For now, we will set both to `center`.

Typically it is not good practice to have your grid and containing objects be different sizes. If done this way, you will need to keep track of main/cross axes for each individual grid cell. You can avoid this by not specifying height and width on your boxes, and not using `justify-self` to align them center - this way they will simply take up the entire space of each grid cell they are put into. You can control sizes and spacing from the parent element by defining your cell sizes and spacing accordingly.

Automatic Columns and Rows for Infinite Grids

There are certain situations where it is possible that you won't know the number of elements to be added to your grid ahead of time. Often times this is the case when returning a search from a database or other such situations.

To account for this, you can add the `grid-auto-rows` property. This adds more rows to your grid as more elements get added to your grid. This is also why manually adding rows using `grid-template-rows` isn't necessary - as long as you've set the proper columns, the rows will align automatically.

While it is possible to set your `grid-auto-rows` to a fraction similarly to your columns, this isn't necessarily good practice, as your grid cell size will be inconsistent as the number of items changes. Instead it's best to establish a hard pixel size such as `300px`. You only need to establish this once, as this is the value that will be pulled each time CSS has to make a new row.

Under normal circumstances, new elements will be added to the grid as new rows. This is due to the `grid-auto-flow` property, which is set to `row` by default. If you instead set this to `column`, it will add new elements of your grid as new columns instead

Typically If you know how many elements will be in your grid, and that number is not likely to change, use the `grid-template-columns` and `grid-template-rows` properties. If your content is likely to change over time, it's best practice to use `grid-auto-columns` and `grid-auto-rows`.

Stretch An Element Across Multiple Cells

A trend in modern CSS and web design right now is the bento layout - a grid of cells that are all different lengths and sizes that is used to present different kinds of information. In order to create one of these bento layouts, you will need to know exactly how many elements will be in your grid, and have an idea in mind for what this grid layout will look like.

In the example of this lesson, we are using a grid with 4 elements. 2 of these elements will be 1 normal grid size each, and the other 2 will stretch across 2 columns. Thus we will need a grid layout of 2 rows and 3 columns to fit all elements. So far, our CSS grid setup looks something like this:

```
body{
    display: grid;
    grid-template-columns: 250px 250px 250px;
    grid-template-rows: 250px 250px;
    gap: 10px;
}
```

To stretch our boxes across multiple cell lengths, we need to understand `grid-column-start` and `grid-column-end`. When we have a grid, there is a certain amount of vertical lines that comprise that grid. In the case of our 3x3 grid, we have *four* lines. These lines are the outermost edges of the grid itself, as well as the two lines in the middle on either side of the central column. When we establish `grid-column-start` and `grid-column-end`, we are establishing at which lines in our grid the specified element will start and end.

For our long boxes, we will need to give them unique IDs in order to set up this bento grid. In addition to their `box` class, I gave them the IDs of `box-1` and `box-4` respectively, allowing them to maintain their `box` styling while giving them additional styling for their IDs. Thus when we give styling to `box-1`, we can establish its start and endpoint:

```
#box-1{
    grid-column-start: 1;
```

```
    grid-column-end: 3;
}
```

This means that our `box-1` will start at the leftmost edge of our grid (line 1) and will end before the last cell begins (line 3). Similarly, styling for `box-4` will look like this:

```
#box-4{
    grid-column-start: 2;
    grid-column-end: 4;
}
```

This means that `box-4` will begin in the second (or central) cell, and will stretch all the way to the rightmost edge of our grid.

It is difficult to describe what this looks like without visual aids, so please accept this ASCII art to visualize what our bento box layout will look like once complete:

```
[ 1 ][2]
[3][ 4 ]
```

Note that you can also use `grid-row-start` and `grid-row-end` to achieve similar results, only vertically across rows. both `grid-row` and `grid-column` have shorthand to allow you to define the start and end of the element on a single line rather than needing to define two properties. This looks like the following:
`grid-row: 1 / 3;` or `grid-column: 2 / 4;`

Where you establish `grid-row` or `grid-column`, then the start and endpoint separated by a front slash.

Creating Complex Bento Layouts With Grid Template Areas

If you are working with a particularly large or complex bento grid, it can be tedious to use `grid-column` and `grid-row` for each individual element. Instead, larger grids should be created with `grid-template-areas`.

Let's say we have 9 divs with simple styling (no height or width set in their selector) and four grid template rows and columns, each 150px.

```
body{
    display: grid;
    grid-template-columns: 150px 150px 150px 150px;
    grid-template-rows: 150px 150px 150px 150px;
    gap: 10px;
```

```
}
.box{
    background-color: #0071ff;
    border-radius: 15px;
}
```

In each of the divs, we need to define a `style` attribute in addition to its class. This will be the `grid-area` style, and each named for the number box that it is. The HTML will look like this:

```
<body>
    <div style="grid-area: box-1" class="box">1</div>
    <div style="grid-area: box-2" class="box">2</div>
    <div style="grid-area: box-3" class="box">3</div>
    <div style="grid-area: box-4" class="box">4</div>
    <div style="grid-area: box-5" class="box">5</div>
    <div style="grid-area: box-6" class="box">6</div>
    <div style="grid-area: box-7" class="box">7</div>
    <div style="grid-area: box-8" class="box">8</div>
    <div style="grid-area: box-9" class="box">9</div>
</body>
```

Now, back in the `body` section of our CSS, we can add our `grid-template-areas` property. This property will contain four strings - one to represent each row. To create a complex bento layout using `0grid-template-areas`, we will define where each of these boxes will go in the rows.

```
body{
    /*In addition to the previous styling*/
    grid-template-areas:
    "box-1 box-2 box-3 box-4"
    "box-1 box-5 box-5 box-4"
    "box-6 box-5 box-5 box-7"
    "box-8 box-8 box-9 box-7";
}
```

If you can visualize what is happening based on these four strings, you can see that we are defining our box shapes on a per-cell basis in these grid areas. It is best to play around with this CSS concept yourself in order to visualize exactly how it works.

CSS Grid Wrapping

It's important to learn how to wrap CSS grids instead of relying purely on flexbox, because using purely flexbox can lead to some pretty ugly resizing of grids if your screen gets to be too

small.

As the example for this lesson, let's establish some HTML and CSS:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Responsive Grid</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>Responsive Grid</h1>
    <div class="grid-container">
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
        </div>
        <div class="card">
            <h2>Lorem Ipsum</h2>
            <p>Lorem text.</p>
```

```
        </div>
    </div>
</body>
</html>
```

```
*{
    margin: 0;
    padding: 0;
}
html{
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    color: white;
    text-align: center;
}
body{
    padding: min(50px, 7%);
    background-color: rgb(13, 13, 20);
}
h1{
    margin: 30px 0;
}
.card{
    padding: 2em;
    border: 1px solid rgb(75, 82, 92);
    border-radius: 10px;
    background-color: #222429;
    text-align: center;
}
```

So we've got several "card" class divs all contained within a `grid-container` div, and some basic styling on everything to give it some polish.

Now we will work on creating the grid and making it responsive. To do this, we'll modify our `.grid-container` and set our display type, columns, rows, etc. If we want to set up a grid that has several columns, it can get pretty tedious writing our desired size over and over. Instead, we can use the `repeat()` function, which takes two arguments: the number of times to be repeated, and the value to repeat. In this case, if we wanted to create 10 columns of 250px each, we could write
`grid-template-columns: repeat(10, 250px);`

Ten columns is more than we need in this case, so we'll set this to `repeat(4, 300px);`. Since we are creating responsive design, it's normally a good idea to stay away from hard-coded size values where possible. However, with flex grids, we can make elements jump to a new row when they fall below the hard-coded size value, so this rigid size is what we want. Thus the

hard-coded value we are wanting to steer clear from is actually the amount of columns - in this case, 4. We can set this value to `auto-fit`, which will automatically generate columns of the desired width as far across the page as possible.

If you're following along, you may note by now that as our items wrap, there is a bit of awkward space to the right of the grid. We can remove this by centering our grid with `justify-content: center;`.

So now we have a grid that wraps adaptively to the screen size, but it would look even better if the individual items also resized themselves based on screen size to always take up the most possible space. Thus we will need to modify our `repeat` value again. Your knee-jerk reaction may be to replace our hard-coded pixel value with `1fr` or similar, but this will serve to only allow a maximum of one column ever, regardless of the screen size. Instead, we need a way to specify a maximum size of 300px per item, and then add more columns if the space allows.

The solution to this problem is a new function called `minmax()`. This allows us to specify a minimum and maximum value. This allows us to provide a size range greater than or equal to `min`, *and* less than or equal to `max`. In other words, the first parameter is our minimum value, and the second parameter is our maximum value.

This allows us to have the best of both worlds - we can set a minimum value of 300px to ensure our items won't shrink below this size, and a maximum value of `1fr` to create new columns as available. To implement this, we will once again modify the `grid-template-columns` property to `repeat(auto-fit, minmax(300px, 1fr));` This will create a grid that ensures each column is at least 300px wide, and will grow bigger when there is enough space.

So in the end, the CSS for our `.grid-container` selector looks like this:

```
.grid-container{
    display: grid;
    border: 1px solid red;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
    gap: 15px;
    justify-content: center;
}
```

Day 6 Side Projects

Create A Responsive Navigation Bar With Sidebar Animation