Learn SASS

https://www.codecademy.com/learn/learn-sass (3hours)

Syntactically Awesome StyleSheets, is an extension language for CSS. With Sass, you can write clean, sustainable CSS code and solve the common repetition and maintenance challenges present in traditional CSS. In addition to being a valuable skill for any front-end developer, transitioning from CSS to Sass is pretty smooth due to the familiar syntax. For this reason, we will be learning the SCSS (**Sassy CSS**) syntax.

Create a SASS Stylesheet

Sass can't be directly interpreted by your browser, so it must first be converted, or compiled, to CSS before the browser can directly understand it.
To compile it to CSS, type the following command in the terminal and pressing enter:
sass main.scss main.css

The sass command above takes in two arguments:
The input (main.scss)
The location of where to place that output (main.css)
Make sure to look closely at the extensions of each file in the command above.

Nesting is the process of placing selectors inside the scope of another selector:
- In programming, a variable's scope is the context in which a variable is defined and available to use.
- In Sass, it's helpful to think of the scope of a selector as any of the code between its opening { and closing } curly brackets.
- Selectors that are nested inside the scope of another selector are referred to as children. The former selector is referred to as the parent. This is just like the relationship observed in HTML elements.

Nesting allows you to see the clear DOM relationship between two selectors while also removing the repetition observed in CSS.

With Sass, you can avoid repeating the parent over and over again and also avoid defining each selector independently.

In SCSS, nesting is not limited only to selectors. You can also nest common CSS properties if you append a : colon suffix after the name of the property.

```scss
.parent {
  color: blue;
  .child {
    font-size: 12px;
  }
}
```

In the example above `.child` is the child selector and `.parent` is the parent selector.

The above SCSS would compile to the following, equivalent CSS:

```css
.parent {
  color: blue;
}

.parent .child {
    font-size: 12px;
}
```

For example, the following SCSS code:

```scss
.parent {
  font : {
    family: Roboto, sans-serif;
    size: 12px;
    decoration: none;
  }
}
```

will compile to the following CSS:

```css
.parent {
  font-family: Roboto, sans-serif;
  font-size: 12px;
  font-decoration: none;
}
```

Variables in SCSS allow you to assign an identifier of your choice to a specific value.
Why is that useful? Unlike in CSS, if you need to tweak a value, you'll only have to update it in one place and the change will be reflected in multiple rules.
In Sass, $ is used to define and reference a variable: $translucent-white: rgba(255,255,255,0.3);
And then you can use it: background-color: $translucent-white;

There are different **data types** you can assign to a variable in CSS.In addition to color, numbers, strings, booleans, and null, Sass also has two other data types, lists and maps.

Lists can be separated by either spaces or commas. For example, the following list denotes font properties, such as: 1.5em Helvetica bold;
or Helvetica, Arial, sans-serif;
So one example : $standard-border: 4px solid black;
Note: You can also surround a list with parentheses and create lists made up of lists.

Maps are very similar to lists, but instead each object is a key-value pair. The typical map looks like:
(key1: value1, key2: value2);
Note: In a map, the value of a key can be a list or another map.

1. **Nesting** is the process of placing child selectors and properties in the scope of a parent selector. This allows a programmer to draw DOM relationships and avoid repetition.
2. **Variables** make it easy to update code and reference values by allowing you to assign an identifier to a value. (start with $)
3. **Sass Data Types** include: Numbers Strings Booleans null Lists Maps

Nesting and variables are just two ways that Sass keeps stylesheets clean.

Mixins and the & Selector

In CSS, a **pseudo-element** is used to style parts of an element, for example:

- Styling the content **::before** or **::after** the content of an element.
- Using a pseudo class such as :**hover** to set the properties of an element when the user's mouse is touching the area of the element.
some CSS selectors (https://www.w3schools.com/cssref/css_selectors.asp)

Insert some text before the content of each <p> element:

```
p::before {
   content: "Read this: ";
}
```

The ::before selector inserts something before the content of each selected element(s).
Use the **content** property to specify the content to insert.
Use the **::after** selector to insert something after the content.
CSS syntax is: `::before {`

```
    css declarations;
}
```

:active	a:active	Selects the active link
::after	p::after	Insert something after the content of each <p> element
::before	p::before	Insert something before the content of each <p> element
:checked	input:checked	Selects every checked <input> element
:disabled	input:disabled	Selects every disabled <input> element
:empty	p:empty	Selects every <p> element that has no children (including text nodes)
:enabled	input:enabled	Selects every enabled <input> element
:first-child	p:first-child	Selects every <p> element that is the first child of its parent
::first-letter	p::first-letter	Selects the first letter of every <p> element
::first-line	p::first-line	Selects the first line of every <p> element
:first-of-type	p:first-of-type	Selects every <p> element that is the first <p> element of its parent
:focus	input:focus	Selects the input element which has focus
:hover	a:hover	Selects links on mouse over
:in-range	input:in-range	Selects input elements with a value within a specified range
:invalid	input:invalid	Selects all input elements with an invalid value
:lang(*language*)	p:lang(it)	Selects every <p> element with a lang attribute equal to "it" (Italian)
:last-child	p:last-child	Selects every <p> element that is the last child of its parent
:last-of-type	p:last-of-type	Selects every <p> element that is the last <p> element of its parent
:link	a:link	Selects all unvisited links
:not(*selector*)	:not(p)	Selects every element that is not a <p> element
:nth-child(*n*)	p:nth-child(2)	Selects every <p> element that is the second child of its parent
:nth-last-child(*n*)	p:nth-last-child(2)	Selects every <p> element that is the second child of its parent, counting from the last child
:nth-last-of-type(*n*)	p:nth-last-of-type(2)	Selects every <p> element that is the second <p> element of its parent, counting from the last child
:nth-of-type(*n*)	p:nth-of-type(2)	Selects every <p> element that is the second <p> element of its parent
:only-of-type	p:only-of-type	Selects every <p> element that is the only <p> element of its parent
:only-child	p:only-child	Selects every <p> element that is the only child of its parent
:optional	input:optional	Selects input elements with no "required" attribute
:out-of-range	input:out-of-range	Selects input elements with a value outside a specified range
:read-only	input:read-only	Selects input elements with the "readonly" attribute specified
:read-write	input:read-write	Selects input elements with the "readonly" attribute NOT specified
:required	input:required	Selects input elements with the "required" attribute specified
:root	:root	Selects the document's root element
::selection	::selection	Selects the portion of an element that is selected by a user
:target	#news:target	Selects the current active #news element (clicked on a URL containing that anchor name)
:valid	input:valid	Selects all input elements with a valid value
:visited	a:visited	Selects all visited links

A CSS **pseudo-element** is a keyword added to a selector that lets you style a specific part of the selected element(s). For example, `::first-line` can be used to style the first line of a paragraph.

In contrast to pseudo-elements, `pseudo-classes` can be used to style an element based on its *state*.

Syntax

```
selector::pseudo-element {
  property: value;
}
```

You can use only one pseudo-element in a selector. It must appear after the simple selectors in the statement.

> **Note:** As a rule, double colons (`::`) should be used instead of a single colon (`:`). This distinguishes pseudo-classes from pseudo-elements. However, since this distinction was not present in older versions of the W3C spec, most browsers support both syntaxes for the sake of compatibility. Note that `::selection` must always start with double colons (`::`).

Index of pseudo-elements

- `::after`
- `::before`
- `::cue`
- `::first-letter`
- `::first-line`
- `::selection`
- `::backdrop`
- `::placeholder`
- `::marker`
- `::spelling-error`
- `::grammar-error`

In Sass, the **&** character is used to specify exactly where a parent selector should be inserted. It also helps write pseudo classes in a much less repetitive way.

For example, the following Sass:

```
.notecard{
  &:hover{
      @include transform (rotatey(-180deg));
    }
  }
```

will compile to the following CSS:

```
.notecard:hover {
  transform: rotatey(-180deg);
}
```

In Sass, a mixin lets you make groups of CSS declarations that you want to reuse throughout your site. The notation for creating a **mixin** is as follows:

```
@mixin backface-visibility {
 backface-visibility: hidden;
-webkit-backface-visibility: hidden;
```

```
  -moz-backface-visibility: hidden;
  -ms-backface-visibility: hidden;
  -o-backface-visibility: hidden;
}
```

Note: **Mixin names and all other Sass identifiers use hyphens and underscores interchangeably**.

```
.notecard {
.front, .back {
    width: 100%;
    height: 100%;
    position: absolute;

    @include backface_visibility;
  }
}
```

s equivalent to the following CSS:

```
.notecard .front, .notecard .back {
  width: 100%;
  height: 100%;
  position: absolute;

   backface-visibility: hidden;
  -webkit-backface-visibility: hidden;
  -moz-backface-visibility: hidden;
  -ms-backface-visibility: hidden;
  -o-backface-visibility: hidden;
}
```

Mixins also have the ability to take in a value. An argument, or parameter, is a value passed to the mixin that will be used inside the mixin, such as $visibility in this example:

```
@mixin backface-visibility($visibility) {
  backface-visibility: $visibility;
  -webkit-backface-visibility: $visibility;
  -moz-backface-visibility: $visibility;
  -ms-backface-visibility: $visibility;
  -o-backface-visibility: $visibility;
}
```

In fact, you should only ever use a mixin if it takes an argument.

The syntax to pass in a value is as follows:

```
@include backface-visibility(hidden);
```

In the code above, hidden is passed in to the backface-visibility mixin, where it will be assigned as the value of its argument, $visibility.

Mixin arguments can be assigned **a default value** in the mixin definition by using a special notation. A default value is assigned to the argument if no value is passed in when the mixin is included. Defining a default value for each argument is optional.

The notation is as follows: `@mixin backface-visibility($visibility: hidden)`

In general, here are **5 important facts about arguments and mixins**:

- Mixins can take multiple arguments.
- Sass allows you to explicitly define each argument in your @include statement.
- When values are explicitly specified you can send them out of order.
- If a mixin definition has a combination of arguments with and without a default value, you should define the ones with no default value first.
- Mixins can be nested.

Here are some concrete examples of the rules:

```
@mixin dashed-border($width, $color: #FFF) {
  border: {
     color: $color;
     width: $width;
     style: dashed;
  }
}

span { //only passes non-default argument
    @include dashed-border(3px);
}

p { //passes both arguments
    @include dashed-border(3px, green);
}

div { //passes out of order but explicitly defined
   @include dashed-border(color: purple, width: 5px);
}
```

In the example above, the color of the border of span elements would be white, the border of paragraph elements would be green, while the div elements would have a thicker purple border.

Sass allows you to pass in multiple arguments in a list or a map format.

For example, take the multiple properties needed to create the college-ruled stripes in the back of our notecard.

```
@mixin stripes($direction, $width-percent, $stripe-color, $stripe-background: #FFF) {
  background: repeating-linear-gradient(
    $direction,
    $stripe-background,
    $stripe-background ($width-percent - 1),
    $stripe-color 1%,
    $stripe-background $width-percent
  );
}
```

In this scenario, it makes sense to create a map with these properties in case we ever want to update or reference them.

```
$college-ruled-style: (
    direction: to bottom,
    width-percent: 15%,
    stripe-color: blue,
    stripe-background: white
);
```

When we include our mixin, we can then pass in these arguments in a map with the following ... notation:

```
.definition {
      width: 100%;
      height: 100%;
      @include stripes($college-ruled-style...); }
```

Note: Remember to always prioritize readability over writing less code. This approach is only useful if you find it adds clarity to your codebase.

In Sass, **string interpolation** is the process of placing a variable string in the middle of two other strings.

In a mixin context, interpolation is handy when you want to make use of variables in selectors or file names. The notation is as follows:

```
@mixin photo-content($file) {
  content: url(#{$file}.jpg); //string interpolation
  object-fit: cover;}

.photo {
  @include photo-content('titanosaur');
  width: 60%;
  margin: 0px auto; }
```

String interpolation would enable the following CSS:

```
.photo {
```

```
  content: url(titanosaur.jpg);
  width: 60%;
  margin: 0px auto;
}
```
Sass allows **& selector** usage inside of mixins. The flow works much like you think it would:

The & selector gets assigned the value of the parent at the point where the mixin is included.
If there is no parent selector, then the value is null and Sass will throw an error.

```
@mixin text-hover($color){
  &:hover {
      color: $color;
  }
}
```
In the above, the value of the parent selector will be assigned based on the parent at the point where it is invoked.

```
  .word { //SCSS:
    display: block;
    text-align: center;
    position: relative;
    top: 40%;
    @include text-hover(red);
  }
```
The above will compile to the following CSS:
```
  .word{
    display: block;
    text-align: center;
    position: relative;
    top: 40%;
  }
  .word:hover{
    color: red;
  }
```
Notice that the mixin inherited the parent selector .word because that was the parent at the point where the mixin was included.

- **Mixins** are a powerful tool that allow you to keep your code DRY (don't repeat yourself). Their ability to take in arguments, assign default values to those arguments, and accept said arguments in whatever format is most readable and convenient for you makes the mixin Sass's most popular directive.
- The **& selector*** is a Sass construct that allows for expressive flexibility by referencing the parent selector when working with CSS psuedo elements and classes.
- **String interpolation** is the glue that allows you to insert a string in the middle of another when it is in a variable format. Its applications vary, but the ability to interpolate is especially useful for passing in file names.

Functions and Operations

Functions and operations in Sass allow for computing and iterating on styles.

With Sass functions you can:
- Operate on color values
- Iterate on lists and maps

- Apply styles based on conditions
- Assign values that result from math operations

The alpha parameter in a color like RGBA or HSLA is a mask denoting opacity. It specifies how one color should be merged with another when the two are on top of each other.

In Sass, the function f**ade-out** makes a color more transparent by taking a number between 0 and 1 and decreasing opacity, or the alpha channel, by that amount:

```
$color: rgba(39, 39, 39, 0.5);
 $amount: 0.1;
 $color2: fade-out($color, $amount);          //rgba(39, 39, 39, 0.4)
```

The **fade-in** color function changes a color by increasing its opacity:

```
$color: rgba(55,7,56, 0.5);
$amount: 0.1;
$color2: fade-in($color, $amount);     //rgba(55,7,56, 0.6)
```

The function **adjust-hue**($color, $degrees) changes the hue of a color by taking color and a number of degrees (usually between -360 degrees and 360 degrees), and rotate the color wheel by that amount.

Sass also allows us to perform mathematical functions to compute measurements— including colors. Here is how Sass computes colors:

The operation is performed on the red, green, and blue components.
It computes the answer by operating on every two digits.
$color: #010203 + #040506;
The above would compute piece-wise as follows:

01 + 04 = 05
02 + 05 = 07
03 + 06 = 09
and compile to: color: #050709;
Sass arithmetic can also compute HSLA and string colors such as red and blue.

The Sass arithmetic operations are: addition + subtraction - multiplication * division /, and modulo %. **SCSS arithmetic requires that the units are compatible**; for example, you can't multiply pixels by ems. Also, just like in regular math, multiplying two units together results in squared units:10px * 10px = 100px * px. Since there is no such thing as squared units in CSS, the above would throw an error. You would need to multiply 10px * 10 in order to obtain 100px.

Each loops in Sass iterate on each of the values on a list. The syntax is as follows:

```
@each $item in $list {
  //some rules and or conditions
}
```

The value of $item is dynamically assigned to the value of the object in the list according to its position and the iteration of the loop.

For loops in Sass can be used to style numerous elements or assigning properties all at once. They work like any for-loop you've seen before.

Sass's for loop syntax is as follows:

```
@for $i from $begin through $end {
  //some rules and or conditions
}
```

In the example above:
$i is just a variable for the index, or position of the element in the list
$begin and $end are placeholders for the start and end points of the loop.
The keywords through and to are interchangeable in Sass

In Sass, **if() is a function** that can only branch one of two ways based on a condition. You can use it inline, to assign the value of a property:

```
width: if( $condition, $value-if-true, $value-if-false);
```

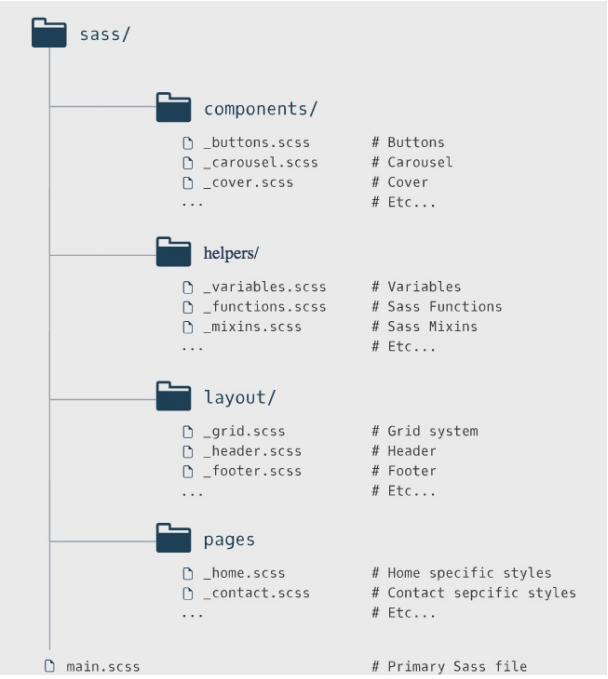
For cases with more than two outcomes, **the @if, @else-if, and @else** directives allow for more flexibility.

```
@mixin deck($suit) {
 @if($suit == hearts || $suit == spades){
   color: blue;
 }
 @else-if($suit == clovers || $suit == diamonds){
   color: red;
 }
 @else{
   //some rule
 }
}
```

The mixin above is a good example for how we would want to handle the coloring of a deck of cards based on a suit condition.

Sustainable SCSS

As your web app or web page grows in complexity, so will the styles that go along with it. It's best to keep code organized.



In addition to having a solid file structure, a big part of staying organized is splitting up the logic into smaller manageable components. **Sass extends the existing CSS @import rule to allow including other SCSS and Sass files.**
-Typically, all imported SCSS files are imported into a main SCSS file which is then combined to make a single CSS output file.
- The main/global SCSS file has access to any variables or mixins defined in its imported files. The `@import command takes a filename to import.

By default, @import looks for a Sass file in the same or otherwise specified directory but there are a few

circumstances where it will behave just like a CSS @import rule:

- If the file's extension is .css.
- If the filename begins with http://.
- If the filename is a url().
- If the @import has any media queries.

In addition to keeping code organized, importing files can also save you from repeating code. For example, if multiple SCSS files reference the same variables, importing a file with variables partial would save the trouble of redefining them each time.

Partials in Sass are the files you split up to organize specific functionality in the codebase.
- They use **a _ prefix notation** in the file name that tells Sass to hold off on compiling the file individually and instead import it. _filename.scss
- To import this partial into the main file - or the file that encapsulates **the important rules and the bulk of the project styles - omit the underscore.**

For example, to import a file named _variables.scss, add the following line of code: @import "variables"; The global file imports all the components and centralizes the logic.

Many times, when styling elements, we want the styles of one class to be applied to another in addition to its own individual styles, so the traditional approach is to give the element both classes.

```
<span class="lemonade"></span>
<span class="lemonade strawberry"></span>
```

This is a potential bug in maintainability because then both classes always have to be included in the HTML in order for the styles to be applied.

Enter Sass's **@extend**. All we have to do is make our strawberry class extend .lemonade and we will no longer have this dilemma.

```
.lemonade {
  border: 1px yellow;
  background-color: #fdd;
}
.strawberry {
  @extend .lemonade;
  border-color: pink;
}
```

If you observe CSS output below, you can see how @extend is working to apply the .lemonade rules to .strawberry:

```
.lemonade, .strawberry {
  border: 1px yellow;
  background-color: #fdd;
}

.strawberry {
  @extend .lemonade;
  border-color: pink;
}
```

If we think of .lemonade as the extendee, and of .strawberry as the extender, we can then think of Sass appending the extender selector to the rule declarations in the extendee definition.

This makes it easy to maintain HTML code by removing the need to have multiple classes on an element.

Sometimes, you may create classes solely for the purpose of extending them and never actually use them inside your HTML.
Sass anticipated this and allows for a special type of selector called a placeholder, which behaves just like a class or id selector, but use the **% notation instead of # or .**

Placeholders prevent rules from being rendered to CSS on their own and only become active once they are extended anywhere an id or class could be extended.

As a general rule of thumb, you should:
- **Try to only create mixins that take in an argument, otherwise you should extend.**
- Always look at your CSS output to make sure your extend is behaving as you intended.

* Sustainability is key in Sass, planning out the structure of your files and sticking to naming conventions for both variables, mixins, and selectors can reduce complexity.

* Understanding CSS output is also essential to writing sustainable SCSS. In order to make the best choices about what functions and directives to use, it is important to first understand how this will translate in CSS.

* Mixins should only be used if they take in an argument, otherwise, you should extend the selector's rules, whether it be a class, id, or placeholder.

In addition to the directives you have learned in this course, be sure to check out the many additional available Sass functions and directives