CSS is a declarative language rather than a programming language and many purists think that it should be kept that way.

However, in my experience I've come across many times when having the power of logical flow control within my styling has been hugely beneficial. Fortunately, Sass can help us out here as it has a number of features that bring the power of programming to CSS.

Loops

Loops are a programming construct that exist in many (if not all) full-on programming languages.

They come in a variety of different forms and are used to execute a block of code (or in the case of Sass, output a block of styles) a number of times.

The way the loop iterates, depends on the type of loop being used. In Sass, there are three kinds of loops:

- the `@for` loop
- the `@while` loop
- and the `@each` loop

Let's look at them in turn and see how they work and what they can be used for.

`@for` loop

A Sass `@for` loop is comprised of 5 parts:

- a counter variable (often written $i short for iteration) which keeps track of how many times the loop has run.
- a starting value for the counter
- an ending value for the counter - when the counter passes this end point the loop will end
- and a keyword taking the value `to` or `through` which determines whether the loop should run up to the end value (stop before it) or through the end value (up to and including it).

- finally there's the body of the loop which contains the styles that will be output. These will use the value of the counter variable in some way such as creating numbered class names or as part of some mathematical calculation.

```
@for $i from 1 through 10 {

    .box-#{$i} {
        width: 10px * $i;
    }

}
```

This example will create 10 class names with a numeric sequence from `.box-1` up to and including `.box-10` where each box has an increasingly larger width; starting at `10px` and ranging up to `100px`.

For loops are very useful when generating custom grid systems with numbered class names for columns of various widths.

`@while` loop

The Sass `@while` loop is a slightly more powerful loop. Instead of having a counter variable that increments by 1 each time, we can control how the counter changes manually.

`@while` loops take the following form and are comprised of 4 key parts:

```
$i: 10;
@while $i > 0 {

    .box-#{$i} {
        width: 10px * $i;
    }
    $i: $i - 2;

}
```

- A starting value of a variable. Here we create a counter variable `$i` with the initial value of `10`.
- A Sass script expression which forms a condition which must be `true` for the loop to execute. In this case, for the style block to be output, `$i` must be greater than `0`.
- The body of the loop where some part of the condition is used.
- A modification to the condition. Here we're decreasing the value of `$i` by 2.

This loop outputs similar code to that of our `@for` loop but the numbered classes are only even numbers and then count down from 10 to 2.

Personally, I use `@while` loops less often than `@for` loops but it's useful to have the option for tighter control of how the loop iterates.

@each loop

The final type of loop available in Sass is the `@each` loop.

The `@each` loop is used to iterate over a collection - a list variable or a map variable for example (we'll look at map variables in detail in the next episode).

A Sass `@each` loop takes the following form:

```
$team: 'rob', 'remy', 'jeff', 'jim';
@each $name in $team {

    .avatar-#{ $name }: {
        background-image: url( "../images/#{ $name }.png" );
    }

}
```

We start with a collection - in this case a list of names.

Then for each item in the list, the loop iterates and stores the value of the current item in the `$name` variable.

In the body of the loop, this `$name` variable is used to create a dynamic class name and output an image with a specific filename for each member of the team.

Each loops are very useful when dealing with lists and wanting to do something specific for each item in the list. We'll look another example of this a bit later on.

But first, let's take a look at another way of controlling how our styles are output.

Conditional Statements

Another concept that exists in most programming languages is conditional statements.

These will only output certain lines of code if a pre-defined condition is true. The condition is a Sass script expression which should evaluate to `true` or `false`.

```
@if ( $is-visible == true ) {
    display:block;
}
```

In this example the `display:block` styles will only be output if the variable `$is-visible` is set to true.

Sometimes we want to output one set of style if the condition is true but other styles in all other cases. Taking the visibility example further, we could add a second part to the conditional to say if it's visible, set `display:block`, else set `display:none`.

```
@if ( $is-visible == true ) {
    display:block;
} @else {
    display:none;
}
```

In this case, only one line of CSS is ever output. Think of it a bit like a fork in the road - each time you approach the fork you have to choose to go one way or the other. `@if` and `@else` conditions work the same way.

Finally, it's possible to set up multiple conditions with `@if` and `@else if` although this can sometimes get long and complex. Here's an example which sets a series of element widths based on a range of variable values:

```
$width: 'large';

@if ( $width == 'small' ) {
    width: 25%;
} @else if ( $width == 'medium' ) {
    width: 50%;
} @else if ( $width == 'large' ) {
    width: 75%;
} @else {
    width: auto;
}
```

As with the other conditional statements we've already seen, only one of these widths will ever be output.

Conditional statements are a very powerful way to control the styles that get output under different conditions and while they may be foreign to styling, they can be incredibly useful when building out complex components or frameworks.

A practical example

Let's wrap up with a practical example.

A few years ago I was tasked with building a live styleguide for a project I was working on. One feature of this styleguide was to show

all the colours used in the project along with it's Sass variable name and it's hex code.

Let's look at a simplified version here and in a future video I'll show you how this can be refactored and made more compact.

Here I've got some static CSS that sets up the styling for a series of boxes for some colour swatches.

The layout is handled with a bit of flexbox to enable easy horizontal and vertical centring of the text inside each box. If you want to dive into flexbox in more detail, check out my course on Flexbox Fundamentals at atozcss.com/courses.

```
.swatches {
    list-style:none;
    margin:1rem 0;
    padding:0;
}
.swatch {
    display:inline-flex;
    justify-content:center;
    align-items:center;
    flex-direction:column;
    width:100px;
    height:100px;
    border:1px solid #000;

    text-align:center;

    &:before { content: ”$color-name” }
    &:after  { content: ”#hex” }
}
```

Each box is going to have a different background colour and show the variable name and corresponding hex code inside.

Let's first set up a series of variables and then create two lists, one for the variable names and one for the list of colours:

```
$names: white, red, green, blue, black;
$colors: #fff, #cc3f85, #9be22d, #66d9ef, #000;
```

We now need to iterate through each name in the list to create a modifier class for each of the colours and output its name.

```
@each $name in $names {
    .swatch--#{ $name } {
        &:before {
            content: ”#{$name}”
        }
```

```
    }
}
```

To add the colour, we need a way to reference the `$colors` variable from within the loop. To do this we can create a counter variable `$i` and use the Sass `nth()` function which returns the nth item in a list.

```
$i: 1;
@each $name in $names {
    .swatch--#{ $name } {
        $color: nth( $colors, $i );
        background: $color;

        &:before {
            content: "#{$name}"
        }
        &:after {
            content: "#{$color}";
        }
    }
    $i: $i + 1;
}
```

We store the color in a variable so it can be used as both the `background-color` and as the `content` property which will display the hex code.

Everything is looking pretty good but we can't read the text in the black box because we have black text on a black background. A quick and dirty solution to this is to check the `lightness` component of the colour and if it's less than `50%`, set the text colour to white.

```
@if lightness( $color ) < 50% {
    color:#fff;
}
```

To browse through the full code for this example, head to atozsass.com/l for the transcript or edit on Codepen.

```
$names: white, red, green, blue, black;
$colors: #fff, #cc3f85, #9be22d, #66d9ef, #000;

.swatches {
    list-style:none;
    margin:1rem 0;
    padding:0;
}
.swatch {
    display:inline-flex;
```

```
    justify-content:center;
    align-items:center;
    flex-direction:column;
    width:100px;
    height:100px;
    border:1px solid #000;

    text-align:center;
}
$i: 1;
@each $name in $names {
    .swatch--#{ $name } {
        $color: nth( $colors, $i );
        background: $color;

        @if lightness( $color ) < 50% {
            color:#fff;
        }
        &:before {
            content: "#{$name}"
        }
        &:after {
            content: "#{$color}";
        }
    }
    $i: $i + 1;
}
```

For a more complex use of loops and conditional statements, and to improve on our quick and dirty method of switching the text colour for readability, we could create a function that measures actual colour contrast instead of just measuring lightness.

This is a little advanced, but I found a great example of such a function by FStop over on github.

CSS may be a declarative language but adding the programming features of loops and conditionals makes Sass an incredibly versatile and powerful extension to the language. And one that I'd be a little lost without for certain situations.