Today we're talking about extending Sass. This is an advanced topic but one that can be very useful when you need to leverage more powerful programming power than Sass alone can provide.

Just to be clear, we're not talking about the Sass `@extend` directive; we covered that in an earlier video in this series. Today we're talking about extending the capabilities of the Sass compiler using the Ruby API.

This is an advanced topic and would ordinarily require some knowledge of the Ruby programming language.

But instead of focusing on programming some complex functionality in Ruby we'll focus on how we actually extend Sass - the specific steps to create and use a custom Ruby function in your styles.

Even if you're not familiar with Ruby you should still be able to follow along and start to see the kind of things that are possible using this technique. First let's have a quick chat about why you might want to do this.

Why Extend Sass with Ruby?

As we've seen throughout this series, Sass is a powerful tool that brings programming features like variables, functions, conditional statements and loops for use in crafting your styles.

There are a number of things that Sass can't do though such as complex math, working with image data or working with environment variables. For certain projects these kinds of things might be useful. In fact, these things are all features of the (now deprecated) Compass extension for Sass.

The Compass library used the same kind of technique as we're going to look at today to add additional functionality to Sass to help with these more complex requirements.

Since Compass is now deprecated you may not want to use it in your projects but might want to bring in certain aspects of its functionality or add similar functionality depending on your specific needs.

Extending Sass with custom Ruby functions is one way to do that. Let's take a look at what's involved.

The Ruby Function

From the docs - Reverse a string. Not very useful but it will allow us to take the example from the Sass documentation as a starting point without having to learn a whole new language.

Here's the function from the Sass documentation:

```
module Sass::Script::Functions
  def reverse(string)
    assert_type string, :String
    Sass::Script::Value::String.new(string.value.reverse)
  end
  declare :reverse, [:string]
end
```

This defines a new function called `reverse` which takes an input `string`, ensure that we are working with a string value and then returns a new Sass string which is the result of reversing the order of the characters in the input string.

That all sounds a bit abstract so let's use a hypothetical example. If I called this new `reverse` function with the string "Sitepoint" the function would return the value "tniopetiS".

There are a couple of other parts to this Ruby snippet worth noting.

Firstly, our custom `reverse` function is defined inside of the `module Sass:Script:Functions`. This is how we are extending the functionality of Sass with our own custom function.

The second is that before ending the snippet we have to `declare` the new `:reverse` function and the arguments it accepts in order for this new function to correctly be added to Sass.

So, we have our new function - how do we use it?

Using New Functionality

To add our new function to Sass we need to create a ruby file. Once created we will run the Sass compiler and add an additional command line flag to include this new file to extend Sass.

I have a simple project here with an assets folder which contains a folder for my `scss` and compiled `css`. The `scss` is very sparse and

just contains a handful of styles that outputs the word “Sitepoint” into the body of the page via a pseudo element.

```
body {
    display:flex;
    height:100vh;
    margin:0;

    &:after {
        content:”Sitepoint”;

        display:block;
        margin:auto;

        color:#339bcb;
        font-size:3em;
        font-family:sans-serif;
    }
}
```

We’re going to extend Sass with our reverse function to reverse the content in this pseudo element.

The first thing to do is create a folder for the new Sass function. I’ll create a `lib` folder within the `assets` folder and within that create a `reverse.rb` file to hold our Ruby function.

Within this file we first need to `require sass` to enable us to extend its functionality. We then add our custom function to the Sass module.

```
require 'sass'

module Sass::Script::Functions
  def reverse(string)
    assert_type string, :String
    Sass::Script::Value::String.new(string.value.reverse)
  end
  declare :reverse, [:string]
end
```

After saving the file we can now update our Sass code to use this new function.

```
&:after {
    content:”#{ reverse( 'Sitepoint' ) }”;
```

This uses string interpolation to add the result of running the reverse function on the string ‘Sitepoint’.

If we try to run the Sass compiler now we don’t get any errors but when our styles compile we see the function call to `reverse` rendered in the

browser. Not ideal.

To complete the process of extending Sass we need to include our custom ruby file at compile time.

To do this, head to your terminal and stop the Sass compiler if it's already running. To include our ruby file with the reverse function (or any other custom ruby file) we need to change the way we run the Sass command.

If we wanted to run the Sass compiler once, we'd use a command such as

```
sass assets/scss/style.scss:assets/css/style.css
```

To include our custom function we add a `-r` flag and the path to our file:

```
sass assets/scss/style.scss:assets/css/style.css -r ./assets/lib/reverse.rb
```

This loads our new function into the Sass functions module and our styles compile without error. If we take a look in the browser we should see the content reversed.

While this is a somewhat trivial example, and it's unlikely that you'll want to use this reverse function in all of your future projects, this concept of extending Sass is a really interesting one. The Sass language is already incredibly powerful but if you want or need to full power of a programming language like Ruby to perform complex functionality then this could open up some interesting possibilities.