7 Functions and Mix-Ins

Contents

[What is a Variable? 1](#_Toc163559085)

[Return or to not Return that is the question 2](#_Toc163559086)

[What is an Argument? 4](#_Toc163559087)

[Seeing an argument in action 5](#_Toc163559088)

[What is a Function? 6](#_Toc163559089)

[What is map-get? 8](#_Toc163559090)

[Creating your first SCSS function 8](#_Toc163559091)

[Mix-ins 11](#_Toc163559092)

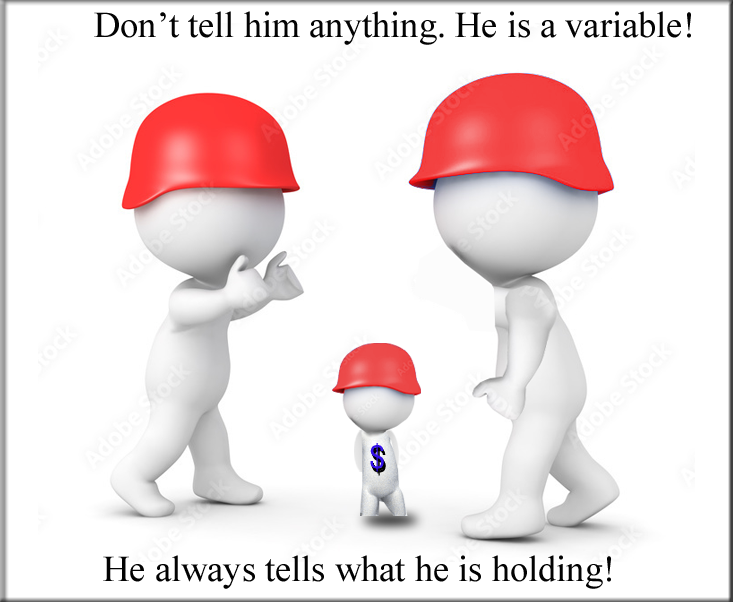
[Creating a Mix-In 13](#_Toc163559093)

[How to include the mixin in your CSS rule 15](#_Toc163559094)

[Mixins Arguments 16](#_Toc163559095)

Make sure you have [watch SASS](file:///C:\xampp\htdocs\Enlightenment\Articles\2024\6-SASS-Or-SCSS\2%20Creating%20the%20Project.docx) on

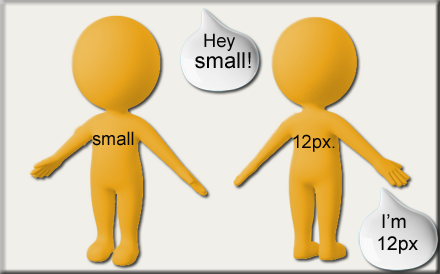
# What is a Variable?

****

In this next illustration we have a variable named small, but when called, it will always tell you it is 12px. At least in this circumstance because that is what we have declared the variable $small to be holding.

small is 12px; that is what I am; call my name, and I will give you just that -- 12px.

$small: 12px;



Basically, if you really think about it, that is what all variables are; they are just place-holders to hold data stuff. So, move over memory, I am pushing in my variables, and now I got some room to put some of my stuff.

# Return or to not Return that is the question

The **return** statement can be confusing when presented in the function. Sometimes you will see the return keyword at the end of a function, and sometimes not. But why is it needed, and when is it not needed?



Well, if you are doing any sort of a calculation, then you will need a return statement. For example, if you are adding two numbers together; the program needs to know this information in order to calculate the sum; and so, that answer needs to be returned. In our function in the next illustration, you will see that we are using the map-get function. This is a sort of calculation, or manipulation (or functionality), to map something to a value.

A function is always two parts, you have the definition then you have a section of the code that needs to run this function. The second part will call the function to get the information it needs; but since these two sections of code need to be in connection to one another to communicate, the return key word is used (and needed) to make this connection, and pass information.



The function call would fill in the place holder with “something,” the function definition would then use what was returned with this “something” from the call’s parentheses, and plug it into its equation, to customize its manipulation, for the calling function.



But in the instance of a simple print statement, the return is not needed.



…or anyways, it is very rare if anyone will return a print statement. That is because print is only used to show information to a human, and the program really isn’t taking and using that information for anything at all. So, since it is text that is not needed to run the program, it does not need to pass information back and forth between the calling code and the function’s definition.

Watch it, be aware that your return keyword must always be used at the end of the function. This is because return is a mechanism which will always throw you out of your function, and fling you back into your calling section’s code. If you do not want to be thrown out of your function prematurely then you need to be very aware of where you are placing this key word.

# What is an Argument?

Sometimes it is helpful to include arguments when writing out your function. An argument is a list of variables; placed inside of parenthesis; and following the name of the function.

**@function name** (**$argument1**, **$argument2**, **$argument3**…) {

SCSS rules, inside function block

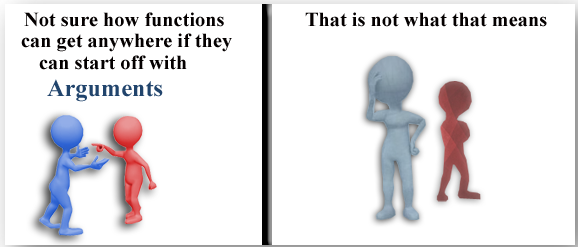
}

If you add an argument to the function’s behavior, it will allow it to be customized individually each time that function is called (needed) to be used somewhere else in the code.



Look at the line of code above, $color is actually an official SCSS variable, and it will represent the color of your font. So, each time you call on this function, and reference this variable, it will already know what to do with @color and how to handle it. Using variables as arguments in your code is great, because they are interchangeable, and can be made to handle very specifically, any code that wants to use it. Even though you might want to create and customize your own variables, using official variables can be a great option; because they come ready-made; straight out of the box, and are there for you to use anytime that you want to use them.

Arguments are great, and throwing them into the parenthesis, after the name of a function, gives the function their own variables to use locally. This means that it just belongs to this function, and will not work outside of it, unless you @return it. The calling code will need access to this information, and so the return keyword gives it this perfect line of communication.



# Seeing an argument in action

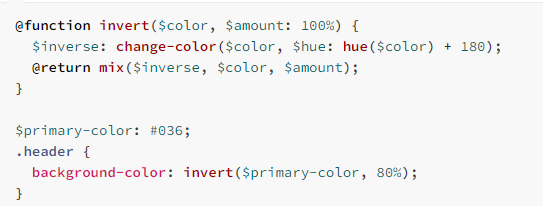
As stated above, $amount is just a variable, that we created to be used in our function. For this example, $amount will take a default value of 100%, as a key/value pair.

We will then be using this $amount value with the hue function. The hue function, like $color variable is official SCSS named code, and so we can just use it, and it will know what to do. Our hue function will be taking our $color variable as an argument. The hue function, comes pre-loaded with a range from 0 to 360 degrees. Because that is how the developers of the SCSS class library determined it to work, when they created this code.

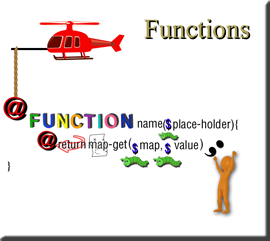
Below you see that hue is taking the variable $color and then adding that amount of 180 degrees to it. This will flip the color in the reverse direction.



Our tutorial next week will dive into this next code a bit deeper. It is just here for reference now.



# What is a Function?



Let’s dive into this function, variable, argument thing just a bit deeper.

**\*Note**-In our illustration above, you will see that our place-holder variable, which is in the parenthesis in the first line, is actually going to be the same thing that will be our value variable, in the second line of our code. So, the place-holder will be filled in with our value, and both the value, and the place-holder values will be the exact same thing.

Functions, in Sass, are very similar to functions that are in JavaScript. So, if you are familiar with JavaScript, you should find this function stuff to be much easier to get the hang of.

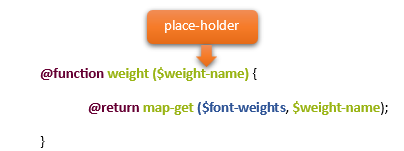
Actually, this function thing is something that you would use if you do not want to keep repeating the same code over and over. You can call a function’s name anywhere in your code, and it is just as if we had written the full code right there in that spot. So, it becomes this type of mechanism that repeats itself all over the place to do its magic. And all you have to do is call it’s name.

Filling in the place holder

Notice how we had put that place-holder in the parenthesis in the first line of our function code in the illustration above. We could have used any name here that we want, but what it is, is a type of place holder. Once we use a variable, as an argument in the function’s first line, we have the space later on in the code to fill in the blank, that it created, with what we want. We also have access to use this variable in the body of the function’s code. It is basically just holding the door open in memory, to put something in there.

In the example below, you will see that we filled in the place-holder with the word regular. This makes sense here, because regular is a type of font-weight, and font-weight is what we mapped our made-up variable of the name of “$weight” to; so now anytime we use this, our SCSS code will know that the font-weight needs to be set to regular. We could just call it like this anywhere in our code, and our function will know to do it magic and turn our font into regular weight setting. How cool is that.

Now what is really awesome is that what is in the parentheses is a variable (\*interchangeable), and you could change it to any of the pre-determined font-weights that you want.



Try changing the word in the parentheses to Bold and watch a whole lot of Bold magic start taking shape with your fonts.



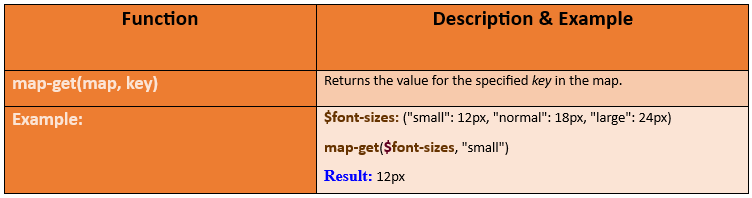
# What is map-get?

By using map-get, in SCSS, we can take several values, and then throw them into a single variable. You can kind of think of these parenthesis, as a way to wrangle in your multiple values, and put them into a group. When the program sees these parenthesis with stuff in them, it knows this stuff goes together in some way.

So, inside of these parenthesis we are using what we want to map, and its value. These are called key: value pairs, at times this is referred to as map,key, which can certainly be confusing, to say the least. But anyways, to continue, the map, key is inside of parenthesis, so we know this stuff goes together. We separate these pairs, inside of those parenthesis with commas.

Notice, in the table below, how we take the value, on the right side of the equation, and throw it into the map on the left side. Look at our variable $font-sizes. This means that anytime we say “small” it will mean 12px. normal will stipulate 18px, and large will specify 24px.

Take this for instance:



# Creating your first SCSS function

Go to your **main.scss** file

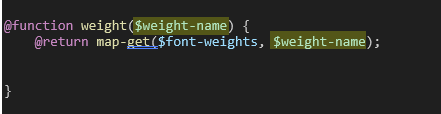
We are going to be placing this function right under our import statements on the page that we have been working with in previous tutorials.

In our example, the $weight-name is where we are going to place the value. Since this is a font weight, our value will be either regular or bold.

$weight-name as a variable is being used as a place holder for either bold or regular, in the map-get function. We will be filling in (bold/regular) when we call it later on in the code. By setting the argument to be $weight-name in the parenthesis, it also initializes it (so-to-speak) and enables us now to use our new variable inside of the function block.

syntax

**map-get** (**map, value**)



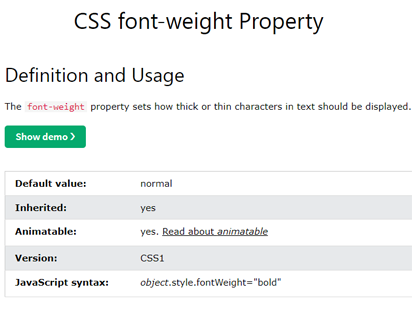
**Notice** how in this code we are replacing $font-weights with the actual official CSS name of font-weight. Then we use a colon, and then we use the function name we created above of “weight.” Inside of the parenthesis is where we fill in our place-holder, with the value of either bold, or regular. You will see from this next example that our function holds all the code now to tell it that the value of regular is what we want to be pulling from font-weight.

Look below, we are starting out with our official name of font weight with a colon after it.



You can find this official CSS in many places but here is the link to the reference by W3schools

<https://www.w3schools.com/cssref/pr_font_weight.php>



Now what is coming after the colon and the official CSS property of font-weight is us using our function name of weight by calling it, and filling in the option of regular. This will show our font at a regular weight. If we filled it in with bold, then it would show our font-weight as bold.



bold



regular



# Mix-ins

Mix-ins are similar to functions.

Go to the **index.html** file and add another paragraph; if you have been following along with our previous tutorials, you should have this file already created.

highlight the entire paragraph on the page, and **alt- shift -down arrow**.

Watch it, do not duplicate the entire div with the class of main, we are only duplicating the paragraph within it.

<div class="main">

<p class="main\_\_p">First paragraph in main.Lorem ipsum dolor, sit amet consectetur adipisicing elit. Temporibus rem voluptatum accusantium itaque atque eum rerum velit veniam, ratione architecto, officiis ab impedit ut assumenda quisquam. Ipsum autem laboriosam illum nobis facere, unde ea, aut nesciunt eius, a voluptas non iste consequatur soluta molestiae doloribus quae esse iusto? Vel, id!</p>

<p class="main\_\_p">First paragraph in main.Lorem ipsum dolor, sit amet consectetur adipisicing elit. Temporibus rem voluptatum accusantium itaque atque eum rerum velit veniam, ratione architecto, officiis ab impedit ut assumenda quisquam. Ipsum autem laboriosam illum nobis facere, unde ea, aut nesciunt eius, a voluptas non iste consequatur soluta molestiae doloribus quae esse iusto? Vel, id!</p>

</div>

Back in the **main.SCSS**.

Add the top 3 lines to your **.main** rule

        display:flex;

        justify-content: center;

        align-items: center;

.main {

        display:flex;

        justify-content: center;

align-items: center;

padding-top: 5%;

        width: 75%;

        margin: 0 auto;

    #{&}\_\_p {

        font-weight:weight(regular);

        &:hover {

            color:chocolate;

        }

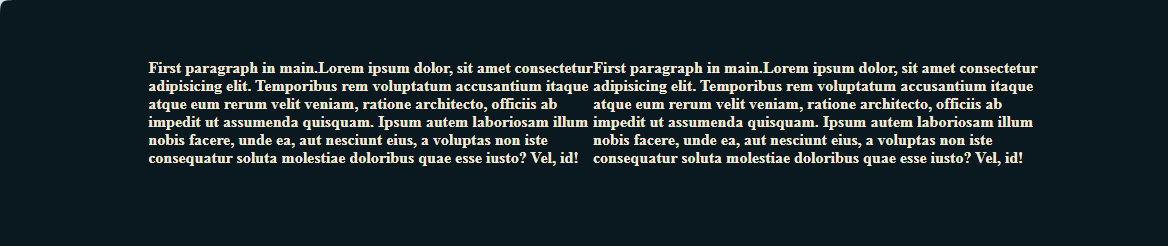
    }

    }

Watch it, remember if you do not have [Watch SASS](file:///C:\xampp\htdocs\Enlightenment\Articles\2024\6-SASS-Or-SCSS\2%20Creating%20the%20Project.docx) on, you will not be able to view this.

**Save** that SASS file and then switch back to the index.html file. And it now is flexible with size of viewport.

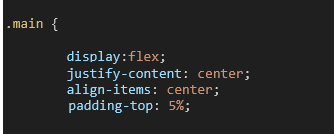
Yes, it looks bad, without a gap between columns, but we will fix that. And you will find that by default, if you have more than one paragraph inside of one div, it will default to columns.

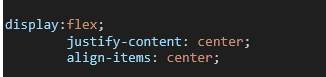


# Creating a Mix-In

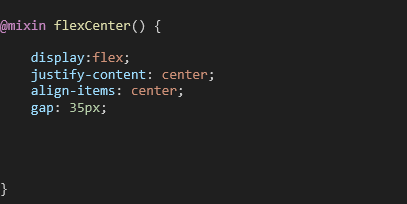


Inside of your main.SCSS file





Cut those top 3 lines from the rule properties that we just added to .main, and put them into our new mixin. We are also adding a gap to make the in-between sections of the paragraphs look better.



You are going to place the mixin before your first tag, which for us is the body tag, and after the maps

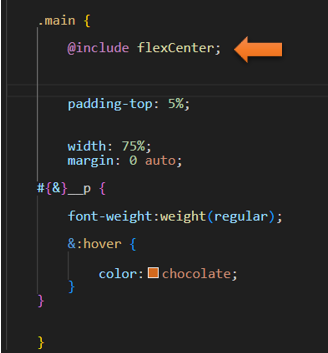
section.



# How to include the mixin in your CSS rule

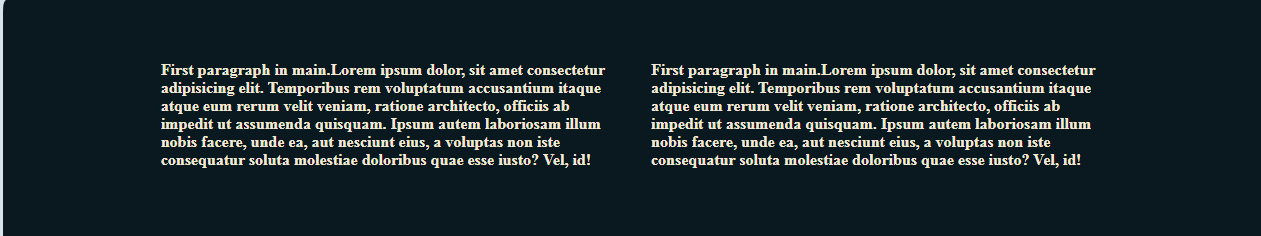
Now that you have defined the mixin, you need to know how to place it into your rule so that our .main can use it.



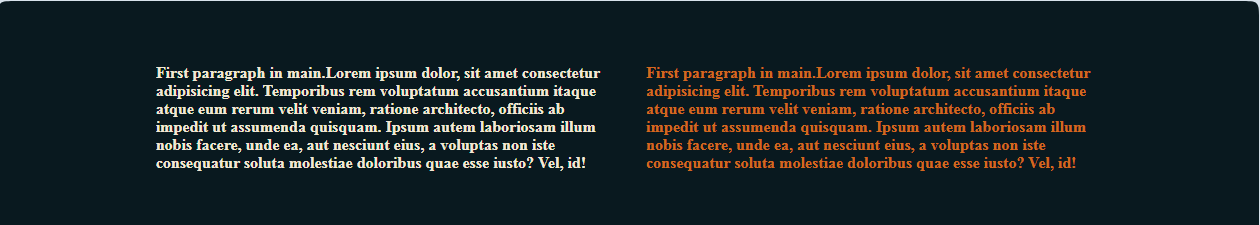


Now, Save it, and go back to index.html, to test the code

… and here you see it is still reading the code correctly!



…and when we hover, you will see that it is also still listening to our code.

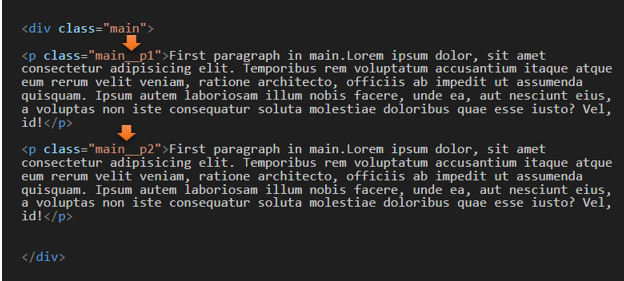


# Mixins Arguments

Remember to [turn on SASS](file:///C:\xampp\htdocs\Enlightenment\Articles\2024\6-SASS-Or-SCSS\2%20Creating%20the%20Project.docx)

Ok, we are going to start out by giving our second paragraph a different class name in the index.html page.

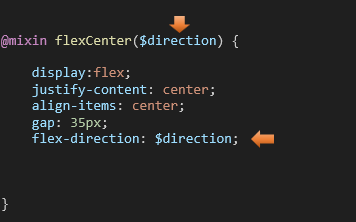
In our index.html file



In Our main. SCSS file

go to our @mixin flexCenter and pass in the variable of $direction

Then inside the rule, add the line that says flex-direction: $direction;

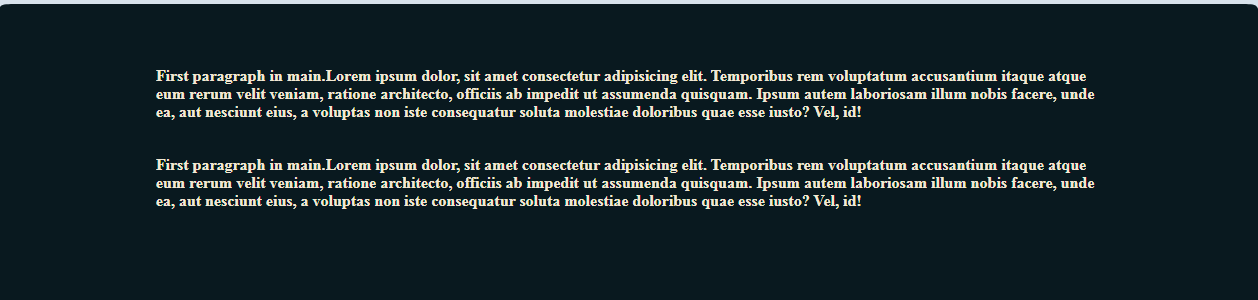


Now go to the rule for .main, and make changes to use this new argument of direction. We add parenthesis and inside, we fill in the blank for $direction. We now want our paragraphs to display in a row, instead of a direction.

column is what we want, this time, for the direction of our paragraphs inside of the div. So, column is what we place inside of the parenthesis for our mixin call for flexCenter.

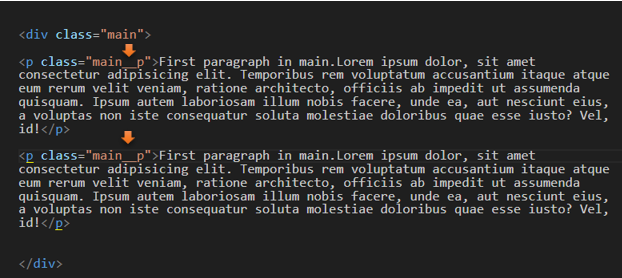


So, now the display is in a column instead of the rows that we had before.



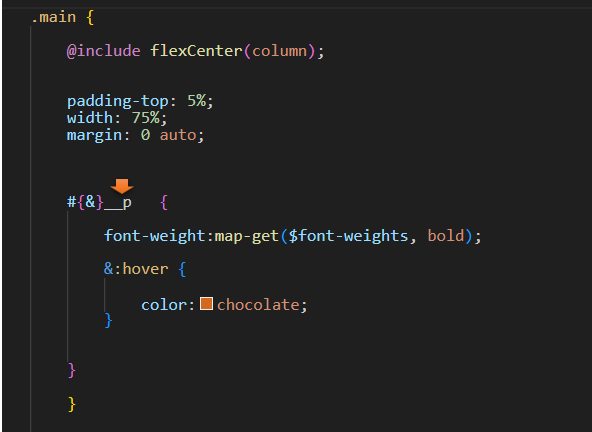
To get the hovers to work again just change those classes on the paragraph tag back to what it was.

In the index.html file. Since our example, that we were trying to demonstrate, is done; we no longer need the two paragraphs to have a separate name. Unless of course, you still require those two paragraphs to listen to transitioning between a row, and a column layout.



In the main. SCSS file

Now our hover will work again



All Better

