7 Functions and Mix-Ins

Contents

[What is a Variable? 1](#_Toc163288324)

[Return or to not Return that is the question 2](#_Toc163288325)

[What is an Argument? 3](#_Toc163288326)

[What is a Function? 4](#_Toc163288327)

[What is map-get? 6](#_Toc163288328)

[Creating your first SCSS function 7](#_Toc163288329)

[Mix-ins 9](#_Toc163288330)

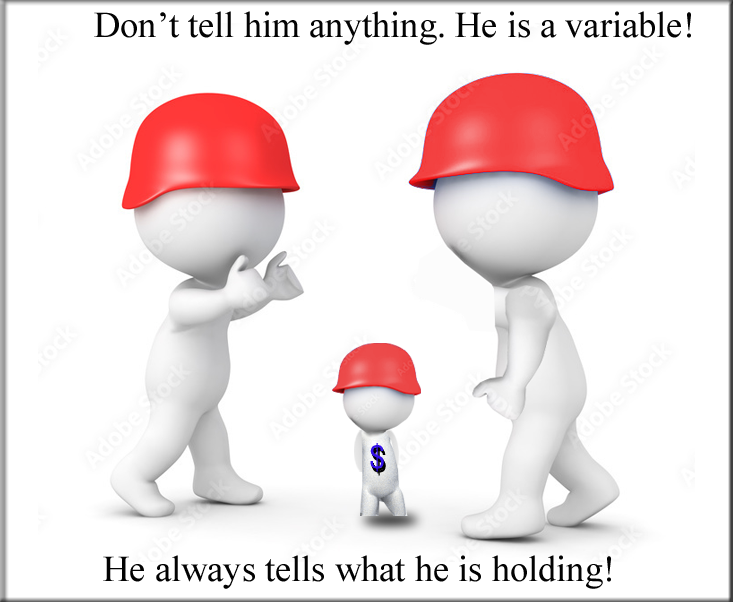
[Creating a Mix-In 11](#_Toc163288331)

[How to include the mixin in your CSS rule 12](#_Toc163288332)

[Mixins Arguments 14](#_Toc163288333)

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

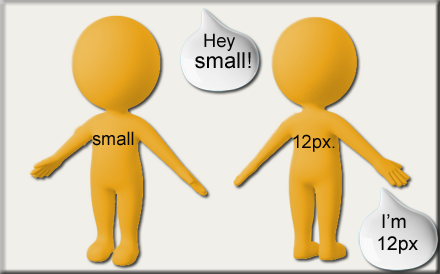
# What is a Variable?

****

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 illustration, you will see that we are using the map-get function. When you call the name function, which is the function that the name function is wrapped in, sometime else in the program, it will need this information. So, this needs to be returned to the calling piece of code.





But you may have noticed in other programing languages that you do not always need this return statement at the end of your function. Take the print statement in Python, 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 be returned to where it is being called.

Also, 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 it.

# 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**…) {

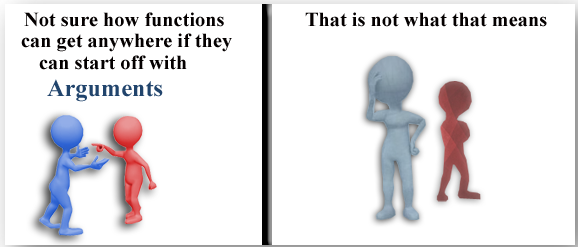
SCSS rules, inside function block

}

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



Look at the line of code above, $color is actually an official SCSS variable, and will represent the color of your text. But since you may not want your text to be the same color each time it is called, you will want to be able to fill this in somewhere else from the calling code. And since it has an @sign in front of it, @color is a variable; a variable is interchangeable, so you can change it to any color you want later in the code.

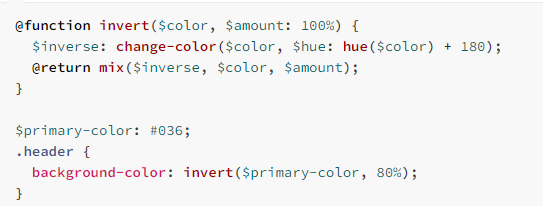


Now the argument **$amount** is a variable that was made up, so we do not have to use variables that were pre-created in the SCSS class library, and just there for us to use. We can also just make them up on the fly, and we told what it will be representing by making it a key:value. We are stating here that 100% will be the default number. We will then be using this $amount value with the hue function, and when the hue function takes it, it will know that its range will be between 0 degrees and 360 degrees. Because that is what hue’s function argument range has already been predetermined to mean by the developers of the SCSS class library when they created this code.

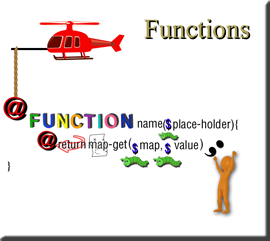
Below you see that $hue as an Official SCSS function is taking the variable $color and then adding that amount of 180 degrees to the hue. This will flip the color in the reverse direction.



Our next tutorial will dive into this code a bit deeper.



# What is a Function?



**\*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; so you see, both the place-holder and the value 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 any where 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.

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 put something in there, we have the space later on in the code to fill in the blank here with what we want. It is basically just holding the door open to put something in it.

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 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.



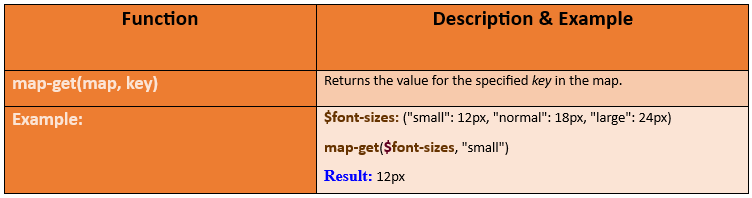
# What is map-get?

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

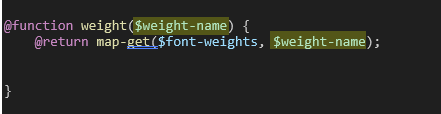
Go to your **main.scss** file

We are going to be placing the functions right under our import statements on the page. In our example, the $weight-name is where we are going to place the property, which we are grabbing out of $font-weights. So, our function name is weight. the word weight is now what we can use to represent this entire block of code thing, when we call it.

$weight-name is used as just a place-holder in the map-get function. We will be filling in what we want to really use here when we call it later in the code. By putting the name $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.

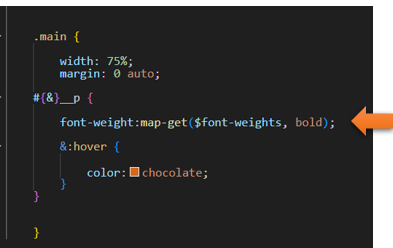
syntax

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



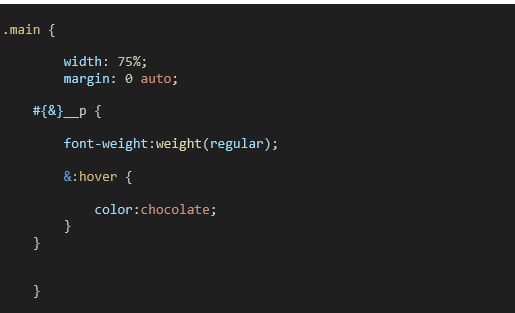
This will be simplifying some of our code, so it does not look so clucky.

Scroll down to where you wrote that nested code. We will be targeting this line of code.



…and with our new function, we can change it to this: We also changed the code from bold to regular, but it is much simpler to write now.

**Notice** how in this code we are replacing $font-weights with the actual CSS name of font-weight. Then we use a colon, and then we use the name of our function. Inside of the parenthesis is where we fill in our place-holder, with the property 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 property of regular is what we want to be pulling from font-weight.



bold



regular



# Mix-ins

Mix-ins are similar to functions.

Go to the **index.html** file and add another paragraph

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 with in 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;

        }

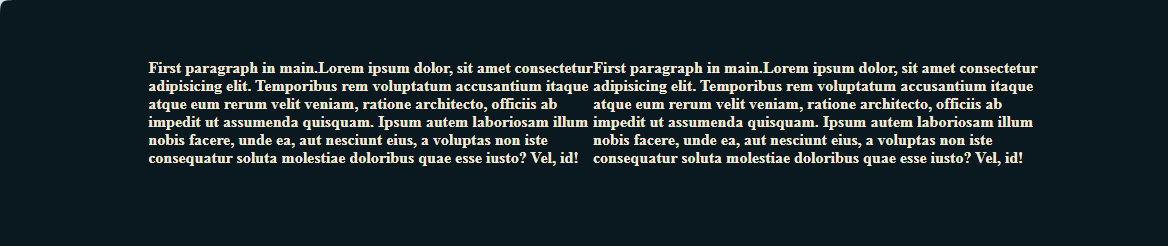
    }

    }

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.

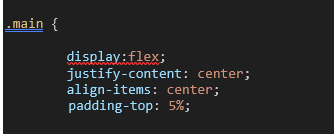
Yes it looks bad, with out a gap between columns but we will fix that. And you will find that by default if you have more than one paragraph inside of one div it will default to columns.



# Creating a Mix-In



Inside of your main.SCSS file



display:flex;

        justify-content: center;

        align-items: center;

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 inbetween sections of the paragraphs look better

@mixin flexCenter() {

    display:flex;

    justify-content: center;

    align-items: center;

    gap: 35px;

}

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

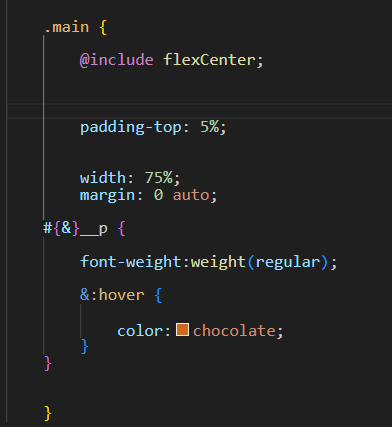
z



# How to include the mixin in your CSS rule

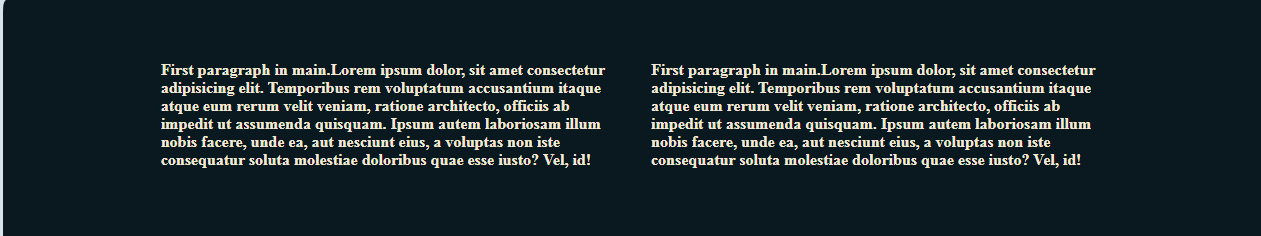
Now that you have defined the mixin, you need to know how to place it into your rule so that our .main can use it.

@include flexCenter;

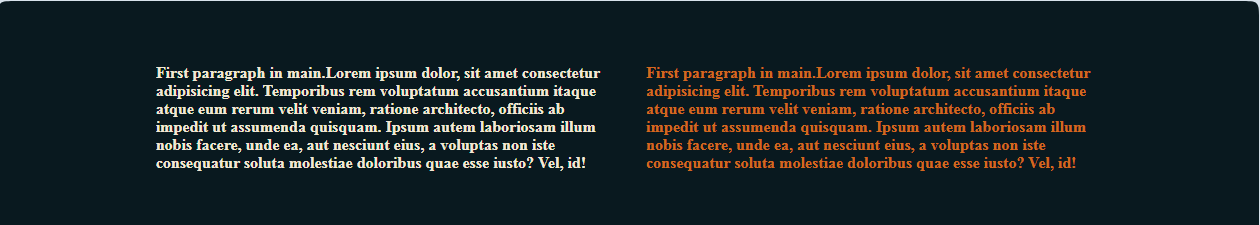


Now, Save it and go back to index.html, to test the code

… and it is still reading the code correctly!



and when we hover…

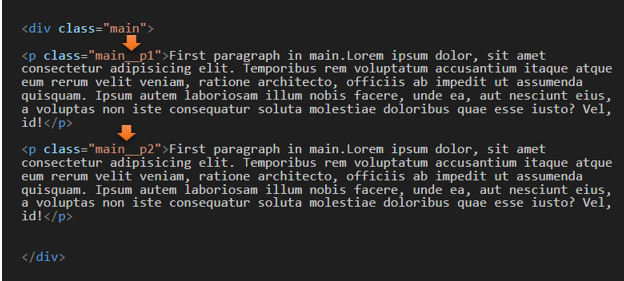


# Mixins Arguments

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;

@mixin flexCenter($direction) {

    display:flex;

    justify-content: center;

    align-items: center;

    gap: 35px;

    flex-direction: $direction;

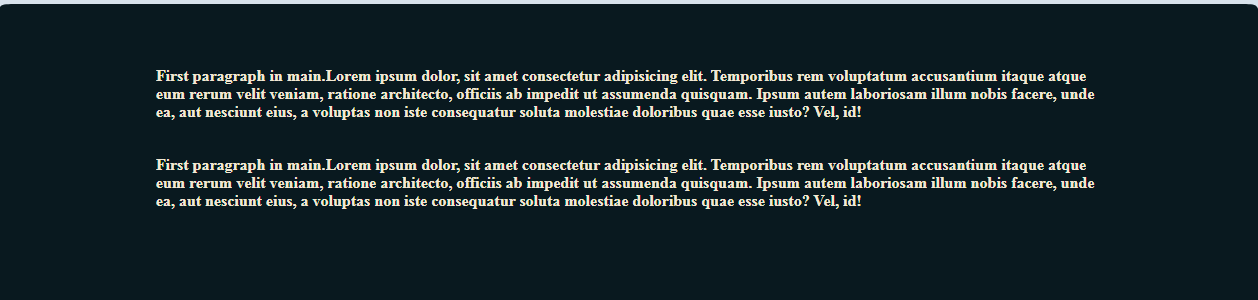
}

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.

.main {

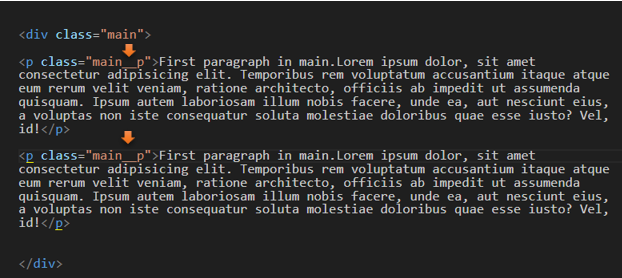
        @include flexCenter(column);

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



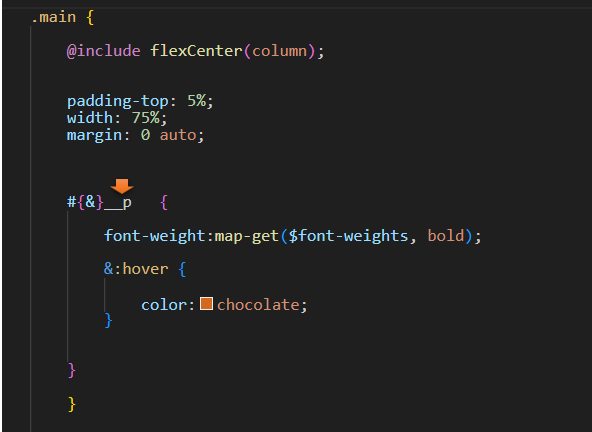
To get the hovers to work again just change those classes on the paragraph tag back to what it was

In the index.html file



In the main. SCSS file

Now our hover will work again



All Better

