Functions are a core concept in many programming languages. But creating and using our own custom functions isn't something we can do in normal CSS. However, Sass brings the power of programming to the table and it does allow us to create our own custom functions for performing all sort of different common everyday tasks.

What and Why?

Functions wrap up a named sequence of instructions that generates a value. They often accept input parameters (called arguments) and return back an output value.

Functions provide the blueprint for a repeatable process that takes one value (or multiple values), performs some kind of calculation and then sends back a new value. They are ideal for performing mathematical operations or converting one type of value into another type of value.

```
{% highlight bash %}
@function function-name( $parameters ) {
// do stuff with input $parameters
@return $output-value;
}
{% endhighlight %}
```

Using functions hands over the responsibility of doing complex calculations to a machine rather than leaving that work and mental overhead to ourselves. This can help us work faster and reduces the risk of human error.

If you're not familiar with programming, the concept of functions can be a little abstract to get your head around to begin with so let's look at an example.

Creating and Using Functions

Let's take the example of converting a pixel value into a rem value. These are both CSS lengths that can be set for a number of CSS properties like `font-size` or `padding`.

When working on a responsive project, it's often preferable to use relative units like `rem` instead of `px` so we can build a flexible

system of proportions rather than sizing every last component to absolute widths, heights and spacing values.

While using `rem` values produces more flexible elements, it's often easier to visualize pixel values. So we could create a Sass function that converts pixels into rems, giving us the best of both worlds and removing the need for constantly reaching for a calculator; I hate CSS maths so if I can get a machine to do this for me, all the better!

Before we look at the Sass syntax for creating and using a function, let's look at the calculation that converts pixels to rems.

The base font size is `16px` which is the equivalent to `1rem`. If we wanted to express `32px` in rems we'd divide 32 by 16 and get the result: `2rem`

{% highlight scss %}
32px / 16px = 2
{% endhighlight %}

If we wanted to express `8px` in rems we'd divide 8 by 16 which gives the result 0.5.

{% highlight scss %}
8px / 16px = 0.5
{% endhighlight %}

So the calculation to convert a pixel value to a rem value is to divide the pixel value by the base font-size and then multiply this by 1rem. We can create a Sass function to automate this process for us as follows:

{% highlight scss %}
$base-font-size: 16px;

@function px-to-rem($value) {
@return ($value / $base-font-size) * 1rem;
}
{% endhighlight %}

Our function accepts an input which we've called `$value` and sends back a `return` value in rems.

To use this function, we call it as follows:

{% highlight scss %}
.element {
font-size: px-to-rem(20px);
}
{% endhighlight %}

This will convert `20px` to `1.25rem` and compile to

{% highlight scss %}
.element {

```
font-size: 1.25rem;
}
{% endhighlight %}
```

Instead of reaching for a calculator or doing a quick sum in our head, we can leverage the power of Sass to do that for us so we can focus on solving more interesting problems.

Real World Functions

The previous example demonstrated the essence of functions; taking one value and turning it into another. But in a real-world project, we may want or need to make our functions more flexible by accepting different types of input.

In a future episode we'll look at adding flow control and conditional statements to functions as well as debugging functions and warning the user of incorrect use, but for now, let's add a couple of extra features to this pixel to rem converter.

Let's make our function more flexible and allow it to accept either a pixel length or a number (that's assumed to be a number of pixels) and convert it into rems.

To do that, we'll first make another function which turns any length value into a number without any units. Once we've done that, we can combine the two together.

To create a function that strips units from a length value, we need to divide the length by a single unit of itself.

So to convert 20px into the number 20, we need to divide 20px by 1px. To convert 80em into the number 80, we need to divide 80em by 1em.

The following function does exactly this:

```
{% highlight scss %}
@function strip-units( $length ) {
@return $length / ( ( $length * 0 ) + 1 );
}
{% endhighlight %}
```

Now we can update our `px-to-rem` function to strip any units from the input values by calling the `strip-units` function.

```
{% highlight scss %}
@function px-to-rem( $value ) {
$value: strip-units( $value );
$base-font-size: strip-units( $base-font-size );

@return ( $value / $base-font-size ) * 1rem;
```

}
{% endhighlight %}

Now we can call the function with a `px` length or a number as follows:

{% highlight scss %}
.element {
font-size: px-to-rem(20px);
padding: px-to-rem(15);
}
{% endhighlight %}

Which will output

{% highlight scss %}
.element {
font-size: 1.25rem;
padding: 0.9375rem;
}
{% endhighlight %}

So not only is our function flexible but it's really useful; instead of seeing values like `0.9375rem` in the stylesheet and trying to get a sense of what that means, we can see a number like `15` and can almost immediately get a sense of the kind of length we're talking about.