GitBook

Sass in the Real World

Book II of IV

Kianosh Pourian and Dale Sande

Table of Contents

1. Introduction
2. BYOF
   i. Anatomy of a function
   ii. Build your own function
   iii. Grids and custom functions
   iv. Function v. Mixin
3. Control Directives
   i. @if control directive
   ii. @for loops
   iii. @while loops
   iv. @each loops
4. Native Sass Functions
   i. Color
   ii. Opacity
   iii. Data
   iv. Introspection
5. Working with Lists
   i. A brief look at font-awesome
   ii. Improvements with lists
6. Working with maps
   i. Improvements with maps
   ii. Further improvements
7. Appendix A
   i. Color functions
   ii. Opacity functions
   iii. Data functions
      i. String functions
      ii. Number functions
      iii. List functions
      iv. Map functions
   iv. Introspection functions
   v. Miscallaneous functions

Sass in the Real World: Book II of IV

In the first book of our series, we told you about our excitement for Sass CSS, the technology behind it and how it can help us in CSS development. In the second book of our series, we continue on the journey through the world of Sass development and we will pick up from where we left off.

We started with the history of Sass, then file management, Object Oriented CSS (OOCSS), scope management, and ended with mixins. These are rules and guidelines that we follow when starting any development project involving Sass. Now with the second part of the series, we will dive deeper into Sass development by learning about:

- Functions: built-in Sass functions and also custom build your own functions
- Control directives: @if, @for, @each, and @while directives
- Working with lists and maps

We appreciate you joining us for this part of the journey and hope that this part of the series is also useful and informative to you.

-- Kianosh Pourian and Dale Sande

Early release book

Writing a book is hard and takes time. So how do we get this new awesome to you as fast as possible? The answer, EARLY RELEASE BOOK! In this EARLY RELEASE BOOK, read chapter-by-chapter while it's being written and, when finished, you get the final book.

The best part is, when you buy the book you are entitled to FREE UPDATES FOR LIFE! Our friends at GitBook.io have made it easy for us to quickly notify you when ever we publish an updated version of the book. It's like WIN all over the place!

About the book and series

Our initial goal of writing this book was to fill a void in Sass CSS books which is a book that covered beyond the basics of Sass CSS development. To fulfill this goal, our initial approach is to provide a single book that covered the A to Z of professional Sass CSS development. While this goal has not changed, our approach has made a small "pivot" (word du jour in today's technology world).

We have divided our single book approach into a 4 part book series. This was done to achieve several goals:

- Be able to publish a book quicker and bringing it into the market faster
- Allow users to select the desired part of the series without having to purchase the entire series. This was a very important part of our approach, as we have seen that different developers are at different levels of Sass CSS learning and development. This will give all a chance to fill in the gaps as needed.

The four part series consist of the following parts:

- *Part 1: Getting Started with Sass*. This part of the series concentrates on the basics of Sass development however with a deeper context and history behind all that is Sass. Our goal for this part of the series was to not only review the basics but also present an explanation behind all the decisions that was made and decisions that a developer must consider and make when developing with Sass
- *Part 2: Deeper Dive*. A continuation of our philosophy on not only understanding why certain structure has been built but also the deeper underlying structure behind it. In this part of the series, we have taken a deper dive into functions (both out-of-the-box functions and custom functions), when to use functions vs. mixins and other tools and processes that accompany Sass CSS development.
- *Part 3: Getting Really Sass'y*. In this part of the series, we continue with all the development needs by talking about issues like responsive design, testing, debugging, and working with frameworks like Zurb Foundation or Twitter

Bootstrap. We will also talk a bit about Compass, the Sass framework that can be accompnied with Sass CSS development.
- *Part 4: Sass in the Stack*. We end the book series with the implementation of Sass in different technology stacks like NodeJS implementation or the Rails asset pipeline. We will also touch upon some performance issues and how to handle these issues.

We hope you enjoy these books, either through individual series or the entire four part series, and this will help you further in your Sass CSS development.

Assumptions

This is the part that most book will label as **"Who is this book for?"** but since we are very strict in our grammar and refuse to end a sentence in a preposition, we have called it "Assumptions", but the sentiment is the same. These are some of the assumptions that we are making:

- We assume that the reader is familiar with Sass CSS and has install Sass on their development machine. If you have not installed Sass CSS, it is very easy, go to Sass-lang.com.
- Running Sass from the command line is preferred, however feel free to use some like Codekit, Compass.app, Scout, or any other desktop application of your choice.
- For testing of new ideas and experimenting with different versions and/or Sass libraries, we suggest using SassMeister, the leading on-line Sass utility.
- Although we are very opinionated about some of our approaches in Sass CSS development, we are agnostic to the technology stack that is being used and promote the usage of Sass in any environment that is suitable and will meet your needs.

So let's start learning about Sass CSS and develop CSS with its' rightful accompanying tool.

Build Your Own Functions (BYOF)

Sass provides an amazing series of build in functions that are very useful in performing regular day to day functions like: darkening a color or going through a list of values. These functions are outlined in the Sass Functions API and we have also included it in the appendix of this book.

However, there are times when a custom function are needed. This is where the power of Sass and the custom function becomes very handy. In the following chapter, we will be talking about how to create a custom function and how to use it in your Sass CSS.

Anatomy of a function

A function is simply a processing machine that will take an input and massage/manipulate/calculate and return the result as an output. Wikipedia defines it as:

> In computer programming, a subroutine is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.*
>
> In different programming languages a subroutine may be called a procedure, a function, a routine, a method, or a subprogram. The generic term callable unit is sometimes used. [1]

In the case of Sass, the goal of a function remains the same. The manipulation of input value(s) resulting in the return of the desired value. A simple example of a function is the following 'em' calculator:

```
$default-browser-font-size: 16px !global;

// A function that calculates the em value of a pixel
// size based on the default font size for the browser
// or the root element

@function emCalculator($size, $context: $default-browser-font-size) {
  @return $size / $context * 1em;
}
```

The general anatomy of a function in Sass is as follows:

A function always begins with `@function` keyword. Parameter(s) for the mixin are not required but it will make the mixin more dynamic if added.

```
@function [function-name]([function-parameters...]) { ... }
```

A function can take advantage of `@if` `@else` , `@for` , `@each` , or `@while` . Also there can be usage of other functions or mixins within the function.

The end result of the function is a return value in the form of a string or number or variable.

```
@return [result];
```

Combined, an example function would look like the following:

```
@function [function-name]([function-parameters ...]) {
  [function logic/Sass functions]
  @return [result]
}
```

As you can see, the `em` calculator function that we created follows along this structure that we have laid out. One rule to follow is to name the function in a descriptive manner. A vague and shortened name will not help other developers understand the reason for the function and as a result it may go unused or misused.

Sometimes the name of a function has been shortened in order to facilitate typing of the function at every use. To help in this matter, we suggest that an overloaded function always accompany the original function that will have a more reasonable name.

```
@function em($size, $context: $default-browser-font-size) {
  @return emCalculator($size, $context);
}
```

In the case of our example, we created the `emCalculator` function and also the overloaded `em` function. The usage of this function will look something like this:

```
.container {
  font-size: em(12px);
}
```

The above code will compile into the following CSS:

```
.container {
  font-size: 0.75em;
}
```

Here is a more complex function example:

```
// ===== Linear gradient color processing =================
//  Function that will process the given number of colors
//  and return the correct color strings as comma separated values
// =====================================================
@function linearGradientColors($stop-colors...) {
  $full: false;
  @each $stop-color in $stop-colors {
    @if $full {
      $full: $full + ',' + $stop-color;
    } @else {
      $full: $stop-color;
    }
  }

  $full: unquote($full);

  @return $full;
}

@function lgc($stop-colors...) {
  @return linearGradientColors($stop-colors...);
}
```

This function takes a variable arguments [2] and returns a comma separated string of values. This is useful so that we are not restricted bby a certain number of parameters, giving us the freedom to use as many necessary parameters.

---

1. Function definition: http://en.wikipedia.org/wiki/Subroutine
2. Variable arguments: arguments at the end of a mixin or function declaration that take all leftover arguments and package them up as a list. These arguments look just like normal arguments, but are directly followed by `...`

BYOF (Build your own function)

Now that we have explored the anatomy of a function, you will quickly come to the conclusion that have the need to build custom functions. Luckily, Sass allows us to do that and has a plethora of tools to be able to create custom functions.

Functions must be used in areas where there is a need for a series of processes that will result in a desired output. For example, we may need to take a string and reverse it, modify a given list, or make a simple calculation to get the 'em' value of a given `pixel` value. Let's take a look at the anatomy of a function.

The Power of Functions

Functions can be very powerful and useful in the world of Sass and CSS. Repeated calculations or abstracted sub-routines are prime example of where functions can become handy. A simple example of a function is to evaluate the color of a font based on the given background color.

In the following example you will see that we are creating the function of `set-font-color`. This function will take two arguments, `$background-color` and `$default-font-color`.

Within the function I will take the value of `$default-font-color` and reassign it to the variable of `$font-color` within the scoped context of this function. Next I will evaluate the `lightness` of the value assigned to `$background-color` to see if it is greater then `50%`. The lightness function[1] is a build in Sass funciton.

If that value is greater then `50%`, then the value of `$font-color` is set to the inverse of `$default-font-color`. Again, `invert`[2] is a built in Sass function.

What's interesting to note is that we are using the `@if` statement in a single case. It's only if the color passes that test that the value will be updated and then passed to the return. If not, then the original value of `$font-color` will get passed to the `@return` statement.

As a last nugget, we created a short hand version of the function as well, called `sfc`.

```
@function set-font-color($background-color, $default-font-color) {
  $font-color: $default-font-color;

  @if lightness($background-color) > 50% {
    $font-color: invert($default-font-color);
  }

  @return $font-color;
}

@function sfc($background-color, $default-font-color) {
  @return set-font-color($background-color, $default-font-color);
}
```

Now that we have our function, how do we use it? Illustrated in the following code example we did a few things to take note of. There are two selectors, `button` and `.secondary-button`. With the first `button` selector we set our default `background-color` and then for the text `color` we will use our new function to evaluate the value of `$button-default-bg-color` against the value of `$button-default-font-color`.

We do this again with the second selector of `.secondary-button` as to determine the type color for that button as well.

```scss
button {
    @include opacity(100);
    width: auto;
    min-width: 100px;
    height: 30px;
    text-align: center;
    font-size: em(12px);
    line-height: 1;
    margin: em(10px);
    display: inline-block;
    position: relative;
    background-color: $button-default-bg-color;
    color: sfc($button-default-bg-color, $button-default-font-color);
}

.secondary-button {
    background-color: $light-gray;
    color: sfc($light-gray, $button-default-font-color);
}
```

The power of functions can become more evident when there are heavy calculations involved. Grid system calculation or geometric calculations for games are prime examples. Let's look at Bourbon's Neat application and the way that they use functions to calculate the grid system implementation. Before we dive into the code, let's review what we are trying to achieve when we create a grid system in CSS.

---

1. Lightness: link to Lightness function in Sass docs
2. Invert: link to invert sass function

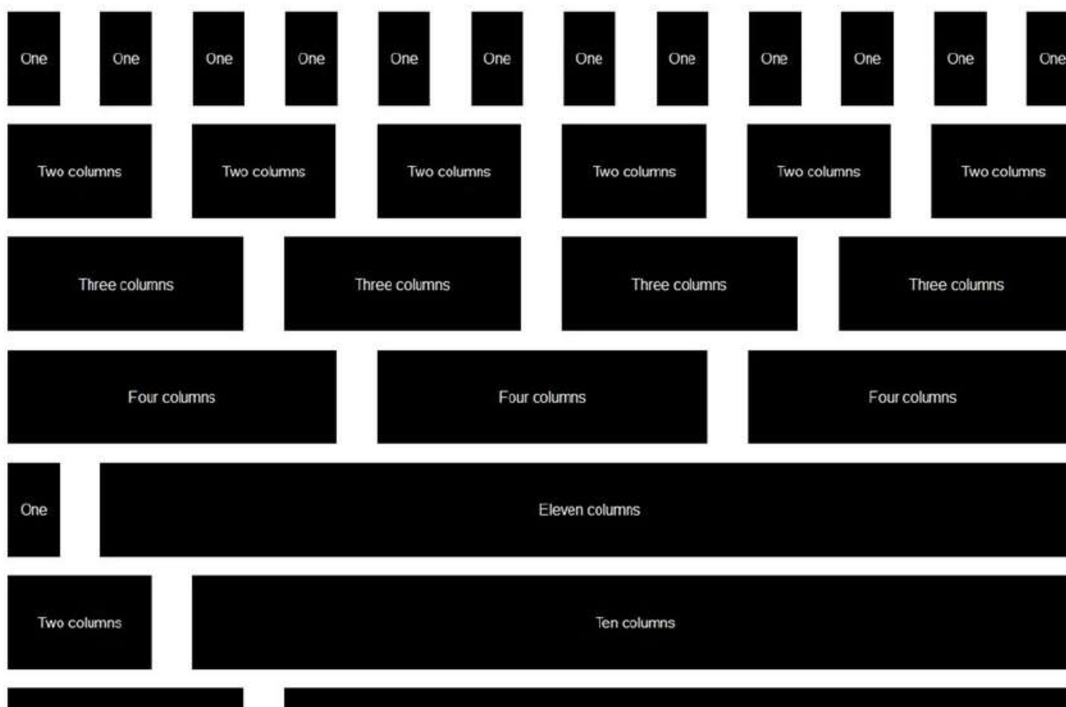
Grid systems and custom functions

In graphic design, a grid is a structure (usually two-dimensional) made up of a series of intersecting straight (vertical, horizontal, and angular) or curved guide lines used to structure content. The grid serves as an armature on which a designer can organize graphic elements (images, glyphs, paragraphs) in a rational, easy to absorb manner. A grid can be use to organize graphic elements in relation to a page, in relation to other graphic elements on the page, or relation to other parts of the same graphic element or shape. [1]

In web design, a grid system is a tool to add order to the chaos of the layout of a web page. The general rules for grid apply with an added dimension of responsiveness to the changing dimensions of a web page. With the grid the page is divided into the determined division components, a 12 or 16 column layout, which will allow for the proper layout of the web page. Each column in the layout will also have an associated gutter, a space between each column, is attached to it. There are two types of grid layout:

1. Fixed: In a fixed grid layout, the parent layout has a fixed width of `960px` or `1024px` for example. All columns within this parent layout will have either a percentage value or a fixed value.
2. Fluid: The difference between fixed and fluid is the fact that the parent layout will have a percentage value in a fluid layout vs. a fixed value in a fixed layout. This fluidity of the grid allows for handling of all the different dimensions that are confronted by web designers and developers, ranging from small devices like a smart-phone to a large display on a big screen or TV.

A typical grid system template looks something like this:



In order to create this grid system, we need to make some calculations in order to come up with the variety of values needed for all the different situations and that is where Sass and the ability to create the necessary custom functions. Let's take a look at how Thoughbot create this grid system in their framework of Bourbon Neat.

Bourbon Neat Functions

In the implementation of the grid system in Bourbon Neat, the following variables have been set in the `_grid.scss` file:

```
$column: golden-ratio(1em, 3) !default; // Column width
$gutter: golden-ratio(1em, 1) !default; // Gutter between each two columns
$grid-columns: 12 !default; // Total number of columns in the grid
$max-width: em(1088) !default; // Max-width of the outer container
$border-box-sizing: true !default; // Makes all elements have a border-box layout
$default-feature: min-width; // Default @media feature for the breakpoint() mixin
$default-layout-direction: LTR !default;
```

The functions that do the majority of the calculations for grid system are in the `_private.scss` file:

```
$parent-columns: $grid-columns !default;
$fg-column: $column;
$fg-gutter: $gutter;
$fg-max-columns: $grid-columns;
$container-display-table: false !default;
$layout-direction: nil !default;

@function flex-grid($columns, $container-columns: $fg-max-columns) {
  $width: $columns * $fg-column + ($columns - 1) * $fg-gutter;
  $container-width: $container-columns * $fg-column + ($container-columns - 1) * $fg-gutter;
  @return percentage($width / $container-width);
}

@function flex-gutter($container-columns: $fg-max-columns, $gutter: $fg-gutter) {
  $container-width: $container-columns * $fg-column + ($container-columns - 1) * $fg-gutter;
  @return percentage($gutter / $container-width);
}

@function grid-width($n) {
  @return $n * $gw-column + ($n - 1) * $gw-gutter;
}

@function get-parent-columns($columns) {
  @if $columns != $grid-columns {
    $parent-columns: $columns !global;
  } @else {
    $parent-columns: $grid-columns !global;
  }

  @return $parent-columns;
}

@function is-display-table($container-is-display-table, $display) {
  $display-table: false;

  @if $container-is-display-table == true {
    $display-table: true;
  } @else if $display == table {
    $display-table: true;
  }

  @return $display-table;
}
```

Let's look at one these functions in detail. The `flex-grid` function calculates the percentage value of each column based on the number of columns and the gutter width.

```
@function flex-grid($columns, $container-columns: $fg-max-columns) {
  $width: $columns * $fg-column + ($columns - 1) * $fg-gutter;
  $container-width: $container-columns * $fg-column + ($container-columns - 1) * $fg-gutter;
  @return percentage($width / $container-width);
}
```

In this functions the parameters passed are `$columns` which is the number of columns that the desired element will be spanning. The value of `$container-columns` is optional which if not passed will be the value of `$fg-max-column` which is 12. In the function, one of the first calculations that happen is to calculate the width of the columns and the container columns. In order to do that, the following calculation is done:

```
$width: $columns * $fg-column + ($columns - 1) * $fg-gutter;
$container-width: $container-columns * $fg-column + ($container-columns - 1) * $fg-gutter;
```

- `$columns` is the value passed to the function.
- `$fg-column` is the value of the variable of $column which itself is the value of the calculation of golden-ratio(1em, 3). For more details on the golden-ratio function go to https://github.com/thoughtbot/bourbon/blob/master/app/assets/stylesheets/functions/_golden-ratio.scss and https://github.com/thoughtbot/bourbon/blob/master/app/assets/stylesheets/functions/_modular-scale.scss
- we also add the width the size of the gutter which equals to the number of `$columns` minus one (because we do not add gutters to all the columns) multiplied by the size of the gutter, `$fg-gutter value`.

the returned value of this function is the ratio of spanning columns to the total number of columns in the parent element represented in percentage. This function is used in the mixin named `span-columns` [2]

```
@mixin span-columns($span: $columns of $container-columns, $display: block) {
  $columns: nth($span, 1);
  $container-columns: container-span($span);

  // Set nesting context (used by shift())
  $parent-columns: get-parent-columns($container-columns) !global;

  $direction: get-direction($layout-direction, $default-layout-direction);
  $opposite-direction: get-opposite-direction($direction);

  $display-table: is-display-table($container-display-table, $display);

  @if $display-table  {
    display: table-cell;
    width: percentage($columns / $container-columns);
  } @else {
    float: #{$opposite-direction};

    @if $display != no-display {
      display: block;
    }

    @if $display == collapse {
      @warn "The 'collapse' argument will be deprecated. Use 'block-collapse' instead."
    }

    @if $display == collapse or $display == block-collapse {
      width: flex-grid($columns, $container-columns) + flex-gutter($container-columns);

      &:last-child {
        width: flex-grid($columns, $container-columns);
      }

    } @else {
      margin-#{$direction}: flex-gutter($container-columns);
      width: flex-grid($columns, $container-columns);

      &:last-child {
        margin-#{$direction}: 0;
      }
    }
  }
}
```

Functions and mixins are powerful tools in Sass. However sometimes the use of one over the other is confused and the mistake is that instead of abstraction, we have a tendency of putting logic in mixins where they belong in a function or vice versa. Let's examine the proper time to use a function and the proper time to use a mixin.

---

1. Grid: http://en.wikipedia.org/)
2. Span Columns: https://github.com/thoughtbot/neat/

Function v. Mixin

In Sass, functions and mixins are very similar. They have the following characteristics in common:

1. They both can access variables or accept arguments
2. They both allow for logic to be applied to the variables and return a certain value

The areas were they divert from each other are:

1. Functions will only return a single value (in any object type supported in Sass; numbers, strings, colors, booleans, lists, and maps)
2. mixins are able to process logic and output CSS rule with attributes and values
3. The `@content` directive cannot be used with functions

Because of their similarities and despite their differences, functions and mixins are sometimes used in an incorrect manner. For example, let's take a look at the following mixin that allows to add a border to a block element:

```
$border-position-all: 'all' !default;
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: #000 !default;

@mixin add-border($border-position: $border-position-all,
 $border-size: $border-default-size, $border-pattern: $border-default-pattern,
 $border-color: $border-default-color) {

 @if $border-position == 'all' {
  border: $border-size $border-pattern $border-color;
 } @else {
  border-#{$border-position}: $border-size $border-pattern $border-color;
 }
}
```

This mixin can be written as function in the following manner:

```
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: #000 !default;

@function add-border-fn($border-size: $border-default-size,
 $border-pattern: $border-default-pattern, $border-color: $border-default-color) {

 $border: $border-size $border-pattern $border-color;

 @return $border;
}
```

This may go under the rule of *"Just because you can doesn't mean that you should"*. The rule that we follow when it comes to implementing an abstracted logic with a custom function or a mixin is the following:

1. Functions should be used for a reusable logic that does a repeated calculation and returns a certain value. For example, the `emCalculator` function that we created earlier in this chapter where it takes a `pixel` value and converts it to `em` values.
2. Mixins should be used for reusable CSS logic, style, or series of properties and values.

Here is an example of a mixin created to handle the variety media query strategies needed for a responsive website:

```scss
// ==== media queries =======================================================
//   EXAMPLE Media Query for Responsive Design.
//   This example overrides the primary ('mobile first') styles
//   Modify as content requires.
//   ================================================================================

//Responsive
//-----------------------------
$small-screen-min-width: em(320px) !default;
$small-screen-max-width: em(767px) !default;
$medium-screen-min-width: em(768px) !default;
$medium-screen-max-width: em(1024px) !default;
$large-screen-min-width: em(1025px) !default;

$screen: "only screen" !default;
$small: "only screen and (min-width:#{$small-screen-min-width}) and (max-width:#{$small-screen-max-width})" !default;
$medium: "only screen and (min-width:#{$medium-screen-min-width}) and (max-width:#{$medium-screen-max-width})" !default;
$large: "only screen and (min-width:#{$large-screen-min-width})" !default;
$landscape: " and (orientation: landscape)" !default;
$portrait: " and (orientation: portrait)" !default;


@mixin respond-to($media, $orientation: false) {
  @if $media == smartphone {
    @if $orientation {
      @if $orientation == landscape {
        @media #{$small} #{$landscape} { @content}
      } @else if $orientation == portrait {
        @media #{$small} #{$portrait} { @content}
      }
    } @else {
      @media #{$small} { @content}
    }
  } @else if $media == tablet {
    @if $orientation {
      @if $orientation == landscape {
        @media #{$medium} #{$landscape} { @content}
      } @else if $orientation == portrait {
        @media #{$medium} #{$portrait} { @content}
      }
    } @else {
      @media #{$medium} { @content}
    }
  } @else if $media == desktop {
    @media #{$large} {@content}
  }
}

// ==== End media queries ====================================================
```

After reviewing this mixin further we have decided to abstracted some of the elements of this mixins, particularly the area where we are handling the logic to build the `@media` label based on the media type that is being passed to the mixin and the function. Here is the mixin, re-written:

```
// ==|== media queries ======================================================
//   EXAMPLE Media Query for Responsive Design.
//   This example overrides the primary ('mobile first') styles
//   Modify as content requires.
//   ==========================================================================

//Responsive
//-----------------------------
$screen: "only screen" !default;
$landscape: " and (orientation: landscape)" !default;
$portrait: " and (orientation: portrait)" !default;

$media-query-sizes: (
  small: (
    min: em(320px),
    max: em(767px)
  ),
  medium: (
    min: em(768px),
    max: em(1024px)
  ),
  large: (
    min: em(1025px)
  )
);

@function media-label($media, $orientation: false) {
  @if(not map-has-key($media-query-sizes, $media)){
    @warn "the $media value needs to be one of the following #{map-keys($media-query-sizes)}";
    @return false;
  }

  $media-sizes: map-get($media-query-sizes, $media);

  $media-label: $screen + " and (min-width:#{map-get($media-sizes, 'min')})";

  @if(length($media-sizes) > 1) {
   $media-label: $media-label +  " and (max-width:#{map-get($media-sizes, 'max')})";
  }

  @if $orientation {
    @if $orientation == landscape {
      $media-label: $media-label + $landscape;
    } @else {
      $media-label: $media-label + $portrait;
    }
  }

  @return $media-label;
}

@mixin respond-to($media, $orientation: false) {
  $media-query-label: media-label($media, $orientation);

  @if $media-query-label {
    @media #{media-label($media, $orientation)} {
      @content
    }
  }
}

// ==== End media queries ===================================================
```

As you can see, we have abstracted the following variables:

```
$small-screen-min-width: em(320px) !default;
$small-screen-max-width: em(767px) !default;
$medium-screen-min-width: em(768px) !default;
$medium-screen-max-width: em(1024px) !default;
$large-screen-min-width: em(1025px) !default;

$screen: "only screen" !default;
$small: "only screen and (min-width:#{$small-screen-min-width}) and (max-width:#{$small-screen-max-width})" !default;
$medium: "only screen and (min-width:#{$medium-screen-min-width}) and (max-width:#{$medium-screen-max-width})" !default;
$large: "only screen and (min-width:#{$large-screen-min-width})" !default;
$landscape: " and (orientation: landscape)" !default;
$portrait: " and (orientation: portrait)" !default;
```

And created a map that contains the breakpoints that we are interested in:

```
$screen: "only screen" !default;
$landscape: " and (orientation: landscape)" !default;
$portrait: " and (orientation: portrait)" !default;

$media-query-sizes: (
  small: (
    min: em(320px),
    max: em(767px)
  ),
  medium: (
    min: em(768px),
    max: em(1024px)
  ),
  large: (
    min: em(1025px)
  )
);
```

We have created a function that will create the media label based on the variables that have been given:

```
@function media-label($media, $orientation: false) {
  @if(not map-has-key($media-query-sizes, $media)){
    @warn "the $media value needs to be one of the following #{map-keys($media-query-sizes)}";
    @return false;
  }

  $media-sizes: map-get($media-query-sizes, $media);

  $media-label: $screen + " and (min-width:#{map-get($media-sizes, 'min')})";

  @if(length($media-sizes) > 1) {
   $media-label: $media-label +  " and (max-width:#{map-get($media-sizes, 'max')})";
  }

  @if $orientation {
    @if $orientation == landscape {
      $media-label: $media-label + $landscape;
    } @else {
      $media-label: $media-label + $portrait;
    }
  }

  @return $media-label;
}
```

This will allow us to re0use this logic in any area that is needed. Our current need is in out `respond-to` mixin:

```
@mixin respond-to($media, $orientation: false) {
  $media-query-label: media-label($media, $orientation);

  @if $media-query-label {
    @media #{media-label($media, $orientation)} {
      @content
    }
  }
}
```

Now we use this mixin wherever we need to add a media query in the following manner:

```
@include respond-to (small) {
  body {
    background-color: red;
  }
}
@include respond-to (small, landscape) {
  body {
    background-color: blue;
  }
}

@include respond-to (medium) {
  body {
    background-color: green;
  }
}

@include respond-to (medium, landscape) {
  body {
    background-color: yellow;
  }
}

@include respond-to (large) {
  body {
    background-color: black;
  }
}

@include respond-to (gubliguke) {// this one will error out and will not be displayed
  body {
    background-color: black;
  }
}
```

The above Sass will compile into the following CSS:

```
@media only screen and (min-width: 20em) and (max-width: 47.9375em) {
  body {
    background-color: red;
  }
}
@media only screen and (min-width: 20em) and (max-width: 47.9375em) and (orientation: landscape) {
  body {
    background-color: blue;
  }
}
@media only screen and (min-width: 48em) and (max-width: 64em) {
  body {
    background-color: green;
  }
}
@media only screen and (min-width: 48em) and (max-width: 64em) and (orientation: landscape) {
  body {
    background-color: yellow;
  }
}
@media only screen and (min-width: 64.0625em) {
  body {
    background-color: black;
  }
}
```

Functions and mixins are the backbone of Sass development and are extremely useful in promoting DRY Sass. Use them wisely and often.

Control directives

Before we dive into more about functions, we will take a step back and take some time to discuss the basic control directives that can be used in some advanced Sass development. In Book I we saw examples some examples of these control directives, but before we talk more about functions, we will cover the core Control Directives in greater detail. Directives covered are:

- @if
- @for
- @each
- @while

These control directives allow for a multitude of logic options designed to handle the different control areas within a function or a mixin.

if(), @if, @else and @else if control directives

Software developer joke:

> A developer, while at the grocery store, calls his wife and asks "Do we need anything from the store?" His wife replies "Yes, we need bread. And if they have eggs, get a dozen" The developer arrives home and to his wife's astonishment, he had bought a dozen loaves of bread.

In the Sass documentation, the `@if` directive is said that it is a SassScript expression that will use the styles nested within the argument if the expression returns anything other then `false` or `null`.

But what does that mean? The ability to evaluate boolean logic is a basics feature of every programming language. This evaluation can involve a basic boolean variable or any other variable/expression that can be evaluated to `true` or `false`.

In the following example we will use the `@if` control directive and a simple boolean variable to evaluate something like this:

```
$truthy: true !default;
$falsy: false !default;

@if $falsy {
  @deubg "I am false";
} @else if $truthy {
  @deubg "I am truth";
}
```

Running this code we get the following output:

```
@deubg "I am truth";
```

But why? What's interesting about this example, the value of `$falsy` is `false`, so this meets the criteria of being either `false` or `null`. As a result, the `@if` statement then reviews the `@else if` query to evaluate it's boolean value. As a result, `$truthy` comes back as `true` so Sass will print out the rules from within. `@else if` works in Sass much like other languages, whereas `@else` will simply return a rule when all other queries have failed and `@else if` must be evaluated and come back as `true` before the rules are used.

If we were to update the variables being set to the following, we would get a different response:

```
$truthy: null !default;
$falsy: true !default;
```

Resolves to:

```
@deubg "I am false";
```

And as long as we are talking about it, the `@debug` directive simply prints the value of a SassScript expression to the error output stream. This is a pretty useful tool when debugging Sass files that contain complicated SassScript, for example:

```
$val: 100;
$var: 1984;

.block {
  @debug $val != $var;
}
```

When processed, Sass will return:

```
<file name>.scss:<lione number> DEBUG: true
```

OR and AND operators

Using `@if` also allows for the use of `OR` or `AND` operators to evaluate multiple logical values. For example, in the following code, we have two variables. One with an empty list and the other with a list containing two objects. The `@if` statement will evaluate if the length of empty list is greater then '`0` OR if the length of $test-list `is greater then` 0`.

```
$empty-list: () !default;
$test-list: (test me) !default;

@if length($empty-list) > 0 or length($test-list) > 0 {
  @debug "I have a list";
} @else {
  @debug "where is my list?";
}
```

Conversely, using the `and` operator, we can evaluate if one thing AND another are true.

```
$empty-list: () !default;
$test-list: (test me) !default;

@if length($empty-list) > 0 and length($test-list) > 0 {
  @debug "I have a list";
} @else {
  @debug "where is my list?";
}
```

The if() function

Much like the `@if` directive, the `if()` function evaluates if something is true or not and then provides the appropriate return. The syntax is pretty simple, as illustrated in the following:

```
if(condition, true, false);
```

But how do we make use of this? Let's say for example we have a need to evaluate the value of a variable and make sure that it is actually a color. Using the `@if` directive, we could write something like the following. Pretty simple really. When using the function of `color()` pass in a color value or something totally off base. The function will use the built in Sass function to evaluate what type the passed in value is. If it is a color, then return the color. If it is not, return a `null` value.

```
@function color($val) {
  @if type-of($val) == color {
    @return $val;
  } @else {
    @return null;
  }
}
```

To use this, simply reference the function with a CSS attribute in this example. Here the function will return a `null` value and thus the CSS rule would not be printed out. **NOTE**, there is a bug in libsass where the `null` value will be printed out.

```
.block {
  color: color(foo);
}
```

Using the new `if()` function, we can actually wrap all this logic up into a single line of code. Remember the syntax?

```
if(condition, true, false);
```

Knowing this, to re-factor the previous code, just do the following:

```
@function color($val) {
  @return if(type-of($val) == color, $val, null);
}
```

loop control directives

Loops or iteration statements are tools in programming that allow for the repeated execution of certain piece of code on a range of values. There are a range of control directives that allow for building of iterative statements:

- @for loop
- @while loop
- @each loop

Here are the details and syntax of each directive.

@for control directive

The `@for` loop directive allows for iteration through a given range of values and executing a certain code within the given range. The benefit of `@for` loops is the ability to specify the range of values. In Sass, the `@for` loop has the following structure:

```
@for $variable from <start-reange> through <end-range>

or

@for $variable from <start-range> to <end-range>
```

Little unknown fact, you can iterate backwards with negative numbers as well. For example:

```
@for $variable from -10 through -1
```

Note the difference between the two statements is the use of `through` vs. `to`. The difference between the two is that the `@for` loop with the `through` keyword will iterate to the given range and include the `end-range` value while the `to` keyword will iterate only to the `end-range`. Let't look at the following example:

```
$class-name: span !default;
$start-width: 25%;

@for $var from 1 through 4 {
   .#{$class-name}-#{$var} {
      width: $start-width * $var;
   }
}
```

The above Sass code will compile to the following:

```
.span-1 {
  width: 25%;
}

.span-2 {
  width: 50%;
}

.span-3 {
  width: 75%;
}

.span-4 {
  width: 100%;
}
```

The same example re-written using the `to` keyword instead of the `through` keyword:

```
$class-name: span !default;
$start-width: 25%;

@for $var from 1 to 4 {
   .#{$class-name}-#{$var} {
      width: $start-width * $var;
   }
}
```

The above Sass code will compile to the following:

```
.span-1 {
  width: 25%;
}

.span-2 {
  width: 50%;
}

.span-3 {
  width: 75%;
}
```

As you can see, when using `through` starting from 5 and ending in 10 will include 1, 2, 3, and 4. While using the `to` keyword starting from 5 and ending in 10 will include 1, 2, 3, but NOT 4.

@for loops and functions

Building your own functions will be a key part of using many of these control directives and @for loops are no exception. As you advance into more complex examples in your project you undoubtedly will start working with lists. In the next section on `@while` loops, we will use a list to influence the output CSS. Depending on how you want to iterate through your list, something you may want to do is reverse the order of the list. Sass does not support this directly, but we can create a custom function using the `@for` directive to address this.

Example

```
@function em($value, $context: 16) {
  @return $value / $context;
}

@function reverse($list) {
   $result: ();

   @for $i from length($list) * -1 through -1 {
      $result: append($result, nth($list, abs($i)));
   }
   @return $result;
}

$list: american virgin delta;
$column: reverse($list);
$length: length($column);

@while $length > 0 {
  .ad-#{nth($column, $length)} {
   font-size: em(10 * $length);
  }
  $length: $length - 1;
}
```

@while control directive

The clinical definition of the `@while` directive is; the `@while` directive takes an expression and while the expression is true, the code block within the `@while` declaration will be implemented.

Let's look at a textbook example:

```
$class-name: span !default;
$start-width: 25%;
$i: 4;

@while $i > 0 {
  .#{$class-name}-#{$i} {
    width: $start-width * $i;
  }
  $i: $i - 1;
}
```

The above code will compile to:

```
.span-4 {
 width: 100%;
}

.span-3 {
 width: 75%;
}

.span-2 {
 width: 50%;
}

.span-1 {
 width: 25%;
}
```

The key with this directive is that you need to set a manual iterator for this to work, for example `$i: $i - 1;`. This iterator will take the original value of `$i` and then subtract `1` with each loop. The directive itself is stating, `@while $i > 0` keep executing the loop. Pretty simple, right?

@each control directive

The `@each` directive iterates through a list of values (for more information on refer to the first book of the series and the chapter on "Core data types") and passes the values into a block of code for execution. A simple example of this type of directive looks something like this:

```
$fruits: apples bananas oranges pomegranates peaches;

@each $fruit in $fruits {
  .i-like-#{$fruit} {
    background-image: url('/images/#{$fruit}.jpg')
  }
}
```

You can also use the `@each` directive with maps. The `@each` directive in combination of maps, there is the ability to extract the key and its corresponding value. Here is an example:

```
$zoo: ("puma": black, "sea-slug": green, "egret": brown);

@each $animal, $color in $zoo {
  .#{$animal}-icon {
    background-color: $color;
  }
}
```

We will tackle more practical examples of these directives and working with lists and maps in the next few chapters.

Native Sass Functions

As you become more familiar with creating custom functions, you undoubtedly will become more familiar with the built-in native functions available with Sass. We have devoted this entire chapter to common real world examples for using these functions in Sass.

What's in the box

Sass comes with a wide array of already built-in functions that are very powerful and in itself can help with most of the commonly used processes. Sass' function list is quite extensive as illustrated by these categories of function types:

- Color functions:
  1. RGB functions
  2. HSL functions
  3. Other color functions
- Opacity functions
- Data functions
  1. String functions
  2. Number functions
  3. List functions
  4. Map functions
- Introspection functions
- Miscellaneous Functions

The functions are very well documented on the Sass CSS function reference guide and also outlined in the appendix for your reference.

Color Functions

Scott Kellum wrote a great sass toolset named Color Schemer which is now part of Team-sass collection. According to the `Color Schemer` README file, the description of the toolset is as such:

> Color schemer is a robust color toolset for Sass. It expands on the existing Sass color functions and adds things like RYB manipulation, set-hue, set-lightness, tint, shade and more. It also leverages these tools adding a full-featured color scheming tool that allows you to set one primary color and create whole color schemes around it.

This toolset takes full advantage of almost all the color functions that comes with Sass.

```
// Changes the hue of a color.
@function ryb-adjust-hue($color, $degrees) {

  // Convert precentag to degrees.
  @if unit($degrees) == "%" {
    $degrees: 360 * ($degrees / 100%);
  }

  // Start at the current hue and loop in the adjustment.
  $hue-adjust: (ryb-hue($color) + $degrees) / 1deg;

  @return hsl(hue(cs-interpolate($hue-adjust)), saturation($color), lightness($color));
}

// Returns the complement of a color.
@function ryb-complement($color) {
  @return ryb-adjust-hue($color, 180deg);
}

// Returns the inverse of a color.
@function ryb-invert($color) {
  @return ryb-adjust-hue(hsl(hue($color), saturation(invert($color)), lightness(invert($color))), 180deg);
}
```

As you can see, the color functions used within several functions that extend the ability to adjust the color scheme not only in RGB and HSL but also in RYB color model.

Opacity Functions

The opacity functions can be used in many areas. One of the best examples of is in color management of buttons and the different states that they can have. Here is an example of a button generator that will build a defualt button and also create its disabled state which y default has a 0.4 opacity:

```
// Button Variables
$blue: #002868 !default;
$lightened-blue: lighten($blue, 15%) !default;
$white: darken(#fff, 4%) !default;
$button-default-bg-color: $blue !default;
$button-default-font-color: $white !default;

%button-props {
  width: auto;
  min-width: 100px;
  height: 30px;
  text-align: center;
  font-size: 1.0em;
  line-height: rs(16px);
  margin: 10px;
  display: inline-block;
  position: relative;
}

@mixin generateButton($bg-color: $button-default-bg-color,
                  $font-color: $button-default-font-color,
                  $disabled-opacity: 0.6) {

     background-color: $bg-color;

      //calculate the lightness of the background color and set the font color
      @if lightness($bg-color) < 80% {
        color: $font-color;
      } @else {
        color: invert($font-color);
      }

    @extend %button-props;

    &:hover {
      background-color: fade-out($bg-color, 0.2);
      cursor: pointer;
    }

    &.button-disabled, &[disabled="disabled"] {
      background-color: fade-out($bg-color, $disabled-opacity);
        cursor: default;
      }
}

button {
    @include generateButton();
}

.action-button {
  @include generateButton(#940404, #EDECC5, 0.6);
}
```

This example will change the opacity of the background color of the button when the user hovers over the button. It also reduces the opacity on disabled buttons. To see the above working example, go to http://codepen.io/kianoshp/pen/zIeaD.

Data functions

As Sass has matured as a language, one of the most explosive areas of the language is the growing support of native data type functions to include strings, numbers, lists, and maps.

In this section we will cover use cases for all these types of functions. But, please be patient. This will most likely be one of the more frequently updated sections of the book as we will be adding new content all the time.

Introspection functions

The list of functions build into this category really can change your perspective on how you engineer your presentational layer code. The ability to test for the availability of a feature, variable, function or mixin is paramount when you start down the path of engineering complex reusable frameworks. It it with these functions we can finally have real tests in our code that can help a user get their work done more efficiently.

But introspection functions don't stop there. Functions that evaluate for the type of a value, or a value's unit for example, really start to show the impressive ability in this category.

Again, we ask for your patience as we begin to explore more and more advanced use cases for these amazing tools.

Working with lists

In this section, we will take practical approach towards using the loop directives that we learned alongside the lists data type available in Sass. We will look at practical example of Sass like the one used in font-awesome and see how we improve on it by using lists and all the other tools available to us.

Icon fonts

Icon fonts are another toolset available to us in UI development which will provide the following benefits:

- Create icons as fonts that will be downloaded as needed
- The ability to change the attributes of an icon just like the ones we can control on fonts. Attributes like size, color, drop-shadow, etc...
- Improved performance by reducing I/O calls on heavy icon images or sprites.
- Able to add icon fonts to buttons, inputs, paragraph, anywhere fonts can be used.
- Better browser support.

Also according to font-aswesome's web site:

> Font Awesome gives you scalable vector icons that can instantly be customized — size, color, drop shadow, and anything that can be done with the power of CSS.

Let's take a look at some of the font-awesome Sass code.

Font-awesome's Sass Code

Font-awesome uses Jeckyll to build most of its assets, which the Sass portion of the product is also taking advantage of it. The file structure is as follows:

```
|- scss/
|--- _bordered-pulled.scss
|--- _core.scss
|--- _extras.scss
|--- _fixed-width.scss
|--- _icons.scss
|--- _larger.scss
|--- _list.scss
|--- _mixins.scss
|--- _path.scss
|--- _rotated-flipped.scss
|--- _spinning.scss
|--- _stacked.scss
|--- _variables.scss
|--- font-awesome.scss
```

The files that we will taking a closer look are `_variables.scss` and `_icons.scss`.

_variables.scss and _icons.scss

`_variables.scss` file initializes all of the variables that show the icons. The file looks like the following:

```
// Variables
// --------------------------

$fa-font-path:        "../fonts" !default;
//$fa-font-path:        "//netdna.bootstrapcdn.com/font-awesome/4.1.0/fonts" !default; // for referencing Bootstrap CDN font files directly
$fa-css-prefix:       fa !default;
$fa-version:          "4.1.0" !default;
$fa-border-color:     #eee !default;
$fa-inverse:          #fff !default;
$fa-li-width:         (30em / 14) !default;

$fa-var-adjust: "\f042";
$fa-var-adn: "\f170";
$fa-var-align-center: "\f037";
$fa-var-align-justify: "\f039";
$fa-var-align-left: "\f036";
$fa-var-align-right: "\f038";
$fa-var-ambulance: "\f0f9";
$fa-var-anchor: "\f13d";
$fa-var-android: "\f17b";
...
```

The `_icons.scss` is the consumer of these variables in the following manner:

```
/* Font Awesome uses the Unicode Private Use Area (PUA) to ensure screen
   readers do not read off random characters that represent icons */

.#{$fa-css-prefix}-glass:before { content: $fa-var-glass; }
.#{$fa-css-prefix}-music:before { content: $fa-var-music; }
.#{$fa-css-prefix}-search:before { content: $fa-var-search; }
.#{$fa-css-prefix}-envelope-o:before { content: $fa-var-envelope-o; }
.#{$fa-css-prefix}-heart:before { content: $fa-var-heart; }
.#{$fa-css-prefix}-star:before { content: $fa-var-star; }
.#{$fa-css-prefix}-star-o:before { content: $fa-var-star-o; }
.#{$fa-css-prefix}-user:before { content: $fa-var-user; }
.#{$fa-css-prefix}-film:before { content: $fa-var-film; }
.#{$fa-css-prefix}-th-large:before { content: $fa-var-th-large; }
.#{$fa-css-prefix}-th:before { content: $fa-var-th; }
...
```

With some of the Sass tools that we have learned, we can better organize and optimize this code.

Improvements with lists

Let's review the code of `_icons.scss` and its repeating pattern:

```
.#{$fa-css-prefix}-glass:before { content: $fa-var-glass; }
.#{$fa-css-prefix}-music:before { content: $fa-var-music; }
.#{$fa-css-prefix}-search:before { content: $fa-var-search; }
.#{$fa-css-prefix}-envelope-o:before { content: $fa-var-envelope-o; }
...
```

The repeating model can be written like this:

```
..#{$fa-css-prefix}-$ICON_VAR:before { content: $ICON-VALUE }
```

Where `$ICON_VAR` are the names set in the repeating pattern in `_variables.scss file` like `$fa-var-adjust: "\f042";`. The `adjust` part of it constitutes the `$ICON_VAR` value. The `$ICON_VALUE` si the value of the set variable which in our example will be `"\f042"`.

Here is where we can use lists to help us out. We can put all the icon names in a list. Let's try it with the above example segment:

```
$icons: glass music search envelope-o;
```

Now that we have the icon names in a list, we can use a `@each` loop to build `_icons.scss`:

```
@each $icon in $icons {
   .#{$fa-css-prefix}-#{$icon}:before {
      ...
   }
}
```

However this list will give only half of the information that we need which is the name of the icons, we also need the value of the icons. We can re-write the list to include the values of the icons also.

```
$icons: (glass "\f000") (music "\f001") (search "\f002") (envelope-o "\f003");
```

What we have done here by using the paraenthesis, we are creating a list of lists. As we loop through the `$icons` list, we will get a list consisting of two items. At this point, we will take advantage of `nth()` (nth function reference) function to extract the icon name and value. So, our `@each` loop can be re-written as such:

```
@each $icon in $icons {
   .#{$fa-css-prefix}-#{nth($icon, 1)}:before {
      content: nth($icon, 2);
   }
}
```

The above setup will give us the same results, but in a better and much more organized manner. However we are not done yet, we can further optimize and organize our code using maps which we will explain in the next section.

Working with maps

The maps data type that was released in Sass 3.3 is an improvement to the existing lists data type. The map data type allows for:

- Key/Value data sets
- Unique list of keys
- Access to the keys and values
- Ability to iterate through these values

Along with this data type, Sass has also included some map functions (which we will be covering in functions: what's in the box) section:

- map-get($map, $key)
- map-merge($map1, $map2)
- map-remove($map, $key)
- map-keys($map)
- map-values($map)
- map-has-key(|$map, $key)
- keywords($args)

Let's take a closer look at how we can use maps by further expanding on the font-awesome example.

Improvements with map

At last turn, we had created a list variable named `$icons` which contained the values needed to create the css for the font-awesome fonts.

```
$icons: (glass "\f000") (music "\f001") (search "\f002") (envelope-o "\f003");
```

With a map data type, we can take advantage of the key/value pair in the following manner:

```
$icons: (glass: "\f000", music: "\f001", search: "\f002", envelope-o: "\f003");
```

To take advantage of this new map structure, our `@each` loop can be re-written as such:

```
@each $icon-name, $icon-value in $icons {
  .#{$fa-css-prefix}-#{$icon-name}:before {
    content: $icon-value;
  }
}
```

As you can see, the first variable `$icon-name` is the key to each pair and the second value `$icon-value` is the value. This is a way of giving us access to the individual key and value. Another way of extracting the key and value is to get them using a single variable. Hence, you can also write the `@each` as such:

```
@each $icon in $icons {
  .#{$fa-css-prefix}-#{nth($icon, 1)}:before {
    content: nth($icon, 2);
  }
}
```

When returning a single variable, the return value is a `list` (data type) of key/value. As you can see, this has made the setup a bit easier. However, we think we can further enhance this by taking further advantage of `maps` and `lists`.

Further improvements

As of writing this book, font-awesome has a total of 439 icons. Also all the Sass variables and setup is within `_variables.scss` and `_icons.scss` which may be hard to maintain. With maps, we can organize these icons further.

Icon categories

The 439 icons, can be divided into 10 categories:

- Web application icons
- File type icons
- Spinner icons
- Form control icons
- Currency icons
- Text editor icons
- Directional icons
- Video player icons
- Brand icons
- Medical icons

Now that we have divided these icons into their category, we can use `maps` to organize them:

Web application icons

```
$fa-web-app-icons-map: (
    adjust: "\f042",
    anchor: "\f13d",
    archive: "\f187",
    arrows: "\f047",
    "arrows-h": "\f07e",
    "arrows-v": "\f07d",
    asterisk: "\f069",
    ...
);
```

File type icons

```
fa-file-type-icons-map: (
    file: "\f15b",
    "file-archive-o": "\f1c6",
    "file-audio-o": "\f1c7",
    "file-code-o": "\f1c9",
    "file-excel-o": "\f1c3",
    "file-image-o": "\f1c5",
    "file-movie-o": "\f1c8",
    "file-o": "\f016",
    "file-pdf-o": "\f1c1",
    "file-photo-o": "\f1c5",
    "file-picture-o": "\f1c5",
    "file-powerpoint-o": "\f1c4",
    "file-sound-o": "\f1c7",
    "file-text": "\f15c",
    "file-text-o": "\f0f6",
    "file-video-o": "\f1c8",
    "file-word-o": "\f1c2",
    "file-zip-o": "\f1c6"
) !default;
```

We will create a map for each category. We can further organize these maps into a `list` like so:

```
$fa-icons-list: $fa-web-app-icons-map,
            $fa-file-type-icons-map,
            $fa-spinner-icons-map,
            $fa-form-control-icons-map,
            $fa-currency-icons-map,
            $fa-text-editor-icons-map,
            $fa-directional-icons-map,
            $fa-brand-icons-map,
            $fa-video-player-icons-map,
            $fa-medical-icons-map;
```

A `list` can take any data type value and in this case we have placed individual `maps` within the `$fa-icons-list`. Now we can iterate through this lists as such:

```
@each $icon-map in $fa-icons-list {
   // this will return the individual maps
}
```

Now that we have the individual maps, we can iterate through the key/value pairs like so:

```
@each $icon-map in $fa-icons-list {
   @each $icon-name, $icon-value in $icon-map {
      .#{$fa-css-prefix}-#{$icon-name}:before { content: #{$icon-value}; }
   }
}
```

With this we have taken advantage of `lists`, `maps`, and control directives.

For the final results, go to the Font-awesome fork and explore it further.

Appendix A

Sass functions

The following is a list of all the functions that is provide by Sass CSS. The following is an API reference of all the out of box functions.

Color Functions

RGB Functions

Function defintion	Function description
rgb($red, $green, $blue)	Combines the values of red, green, and blue to create the desired color.
Example: color: rgb(186, 218, 85); //#bada55	
rgba($red, $green, $blue, $alpha)	Same as previous function (rgb($red, $green, $blue)) with the ability to set opacity value.The opacity must be between 0 and 1 with increments of 0.1 for different opcaity value.This is very similar to existing CSS rgba() function, as matter of fact it will return the same value.
Example: color: rgba(186, 218, 85, 0.5); //#bada55 with 50% opacity	
red($color)	Get the value of the red component of any color. The value retruned is betwen 0 to 255
Example: red(#bada55); //186	
green($color)	Get the value of the green component of any color. The value retruned is betwen 0 to 255
Example: green(#bada55); //218	
blue($color)	Get the value of the blue component of any color. The value retruned is betwen 0 to 255
Example: blue(#bada55); //85	
mix($color1, $color2, $weight: 50%)	Mixes two colors. If the wieght is not given, then the default 50% weight is applied which means that equals amounts of both color is applied. However if the weight is given, in percentage, then the weight is applied to the first color and the remainder is applied to the second color. For example a weight of 15% means that 15% of the first color and 85% of the second color.

Example:
mix(#bada55, #ababab); //#b2c280, slightly grayer color of #bada55
mix(#bada55, #ababab, 75%); //#b6ce6a, only 25% gray added to #bada55	

HSL Functions

Function defintion	Function description
hsl($hue, $saturation, $lightness)	Given a hue value (a value between 0 to 360), saturation value (a value between 0% to 100%), and a lightness value (a value between 0% to 100%); a color is returned

Example:
color: hsl(0, 100%, 50%) //red or #ff0000
color: hsl(120, 75%, 75%) //pastel green or #8fef8f	

hsla($hue, $saturation, $lightness, $alpha)	Same as previous function (hsl($hue, $saturation, $lightness)) with the ability to set opacity value.The opacity must be between 0 and 1 with increments of 0.1 for different opcaity value. This is very similar to existing CSS rgba() function, as matter of fact it will return the same value, which means the return value for this function is the equivalent 'rgba' value.
Example: hsla(55, 24%, 30%, 0.6) //rgba(95, 92, 58, 0.6)	
hue($color)	This function will return the hue value of the given color which is value between 0 and 360 degrees

<table>
<tr><td colspan="2">Example:
hue(#98FB98) //120deg</td></tr>
<tr><td>saturation($color)</td><td>This function will return the saturation value of the given color which is value between 0 to 100%</td></tr>
<tr><td colspan="2">Example:
saturation(#98FB98) //95.52336%</td></tr>
<tr><td>lightness($color)</td><td>This function will return the lightness value of the given color which is a value between 0 to 100%</td></tr>
<tr><td colspan="2">Example:
lightness(#98FB98) //79.01961%</td></tr>
<tr><td>adjust-hue($color, $degrees)</td><td>By adding the degree to the color this function will allow the hue to adjusted accordingly. The value of the '$degree' variable should be an integer between 0 and 360 degree. You can also use negative numbers.</td></tr>
<tr><td colspan="2">Example:
adjust-hue(#98FB98, -90deg) //#FBCA98
adjust-hue(#98FB98, 90deg) //#09C9FB</td></tr>
<tr><td>lighten($color, $amount)</td><td>One of the most used functions are the lighten and darken functions. This function changes the lightness value by adding the given '$amount' variable to the lightness value of the color returning the resulting color.</td></tr>
<tr><td colspan="2">Example:
lighten(#98FB98, 10%) //#C9FDC9</td></tr>
<tr><td>darken($color, $amount)</td><td>This function changes the lightness value by subtracting the given '$amount' variable to the lightness value of the color returning the resulting color.</td></tr>
<tr><td colspan="2">Example:
darken(#98FB98, 10%) //#67F967</td></tr>
<tr><td>saturate($color, $amount)</td><td>This function allows to adjust the saturation value of the given color by increaing the saturation value. The '$amount' variable must be between 0 and 100%.</td></tr>
<tr><td colspan="2">Example:
saturate(#98FB98, 10%) //#94FF94</td></tr>
<tr><td>desaturate($color, $amount)</td><td>This function allows to adjust the saturation value of the given color by decreasing the saturation value. The '$amount' variable must be between 0 and 100%.</td></tr>
<tr><td colspan="2">Example:
desaturate(#98FB98, 10%) //#9DF69D</td></tr>
<tr><td>grayscale($color)</td><td>This function converts the given color to its equivalent grayscale value. You can achieve the same thing by using desaturate($color, 100%)</td></tr>
<tr><td colspan="2">Example:
grayscale(#98FB98) //#CACACA</td></tr>
<tr><td>complement($color)</td><td>All colors have degree value on the color wheel in degrees. This function will return the complementary color by returning the color on the opposite side of the colro wheel of the given color. This is identical to using saturate($color, 180deg).</td></tr>
<tr><td colspan="2">Example:
complement(#98FB98) //#FB98FB</td></tr>
<tr><td>invert($color)</td><td>This function takes the RGB value of a given color and and inverse all the values. Inverting a color follwos this formula:
invert_red = 255 - red.value
invert_green = 255 - green.value
invert_blue = 255 - blue.value
invert_color = rgb(invert_red, invert_green, invert_blue)</td></tr>
<tr><td colspan="2">Example:
invert(#98FB98) //#670467</td></tr>
</table>

Other Color Functions

<table>
<tr><th>Function defintion</th><th>Function description</th></tr>
<tr><td>adjust-color($color, [$red], [$green], [$blue], [$hue], [$saturation], [$lightness], [$alpha])</td><td>This function will allow the adjustment of one more aspect of a color. One can for example cchange the red value of a color or change the saturation or lightness. Keep in mind that you cannot change the RGB value and the HSL value at the same time. The value range of each parameter is as follows:
• $red, $green, $blue: between -255 and 255
• $hue: between -360 and 360 degrees
• $saturation, $lightness: between -100% and %100
• $alpha: between -1 and 1, in 0.1 increments</td></tr>
<tr><td colspan="2">Example:
adjust-color(#98FB98, $red: -5); //#93fb98
adjust-color(#98FB98, $saturation: 45%); //#94ff94</td></tr>
<tr><td>scale-color($color, [$red], [$green], [$blue], [$saturation], [$lightness], [$alpha])</td><td>sclae-color function is different from adjust-color function in the sense that the adjustment that happens to the given color is between the current value and the high end value. For example, let's look at #bada55. It has a red value of 186, green value of 218, and a blue value of 85. If we use scale-color function and scale the blue value by 50%, it will change the blue by 50% between the original value of 85 and high end value of 255. If we scale it by -50%, it will scale it between the original value of 85 and low end value of 0. All the values passed in the parameters are between -100% and 100%.</td></tr>
<tr><td colspan="2">Example:
scale-color(#bada55, $blue: 50%); //#badaaa, which has an RGB value of rgb(186, 218, 170)
scale-color(#bada55, $blue: -50%); //#bada2a, which has an RGB value of rgb(186, 218, 42)</td></tr>
<tr><td>change-color($color, [$red], [$green], [$blue], [$hue], [$saturation], [$lightness], [$alpha])</td><td>This function is similar to adjust-color function.</td></tr>
<tr><td colspan="2">Example:
change-color(#98FB98, $lightness: 45%) //#09dd09;</td></tr>
<tr><td>ie-hex-str($color)</td><td>Converts a color that can be understood by IE filters. In IE, there is the concept of aRGB color as opposed to RGBa color values. The 8 digit color value consists of:
• First two digits are the values for the alpha channel
• The remaining six values are the hexadecimal value of the color

This function converts a regular hexadecimal or RGBa value to aRGB value and returns it as a string value.</td></tr>
<tr><td colspan="2">Example:
ie-hex-str(#bada55); //"#FFBADA55"
ie.hex-str(rgba(0, 255, 0, 0.5)); //#8000FF00</td></tr>
</table>

Opacity Functions

Function defintion	Function description
alpha($color) / opacity($color)	Returns the opacity/alpha value of the color. This value ranges between 0 and 1 with 0.1 increments

Example:
$default-color: rgba(#cfa842, 0.7);
$opacity-value: alpha($default-color); // 0.7
$opacity-value: opacity($default-color); // 0.7	

rgba($color, $alpha)	Refer to color functions
Example:	
opacify($color, $amount)/fade-in($color, $amount)	These two functions increase the opacity value of a color by the given value. It takes a value between 0 and 1 in 0.1 increments.

Example:
opacify(rgba(#cfa842, 0.7), 0.1); //rgba(207, 168, 66, 0.8)
fade-in(rgba(#cfa842, 0.7), 0.2); //rgba(207, 168, 66, 0.9)	

fade-out($color, $amount)/transparentize($color, $amount)	These two functions decrease the opacity value of a color by the given value. It takes a value between 0 and 1 in 0.1 increments.

Example:
transparentize(rgba(#cfa842, 0.7), 0.1); //rgba(207, 168, 66, 0.6)
fade-out(rgba(#cfa842, 0.7), 0.2); //rgba(207, 168, 66, 0.5)	

Data Functions

The data functions in Sass is quite extensive and as a result we have broken them down into their functional groups. The groups are:

- String functions
- Number functions
- List functions
- Map functions

We will get into more detail in the upcoming sections.

String functions

Function defintion	Function description
unquote($string)	Removes quotes from any string that is passed to the function
Example: unquote("Arial"); //Arial	
quote($string)	Adds quotes to any string that is passed to the function
Example: quote(sans-serif); //"sans-serif"	
str-length($string)	Retrieve the length of any string passed to the function
Example: str-length("I am a string with a length of 33"); //33	
str-insert($string, $insert, $index)	This function will inster a string at the given index. If the $index has a positive value then the index will begin from the left side of the $string. If it has a negative value it will begin from the right side of the $string. If the $index value is out of bounds of the $string length then it will add it to the end (if it is a positive value) or it will add to the beginning (if it is a negative value).

Example:
str-insert("abcd", "efg", 5); //"abcdefg"
str-insert("abcd", "efg", 2); //"aefgbcd"
str-insert("abcd", "efg", -2); //"abcefgd"
str-insert("abcd", "efg", 10); //"abcdefg"
str-insert("abcd", "efg", -6); //"efgabcd"	

str-index($string, $substring)	Gives the index of the first occurence of the $substring value within the $string value
Example: str-index("The quick brown fox jumps over the lazy dog", "fox"); //17	
str-slice($string, $start-at, [$end-at])	Returns a new sliced string that matches the $start-at index and goes to the index of the $end-at value. The $end-at value is optional, if not provided then the string is sliced starting from the $start-at point and ending at the end of the string
Example: str-slice("Humpty Dumpty sat on a wall, Humpty Dumpty had a great fall", 0, 27); //Humpty Dumpty sat on a wall str-slice("Humpty Dumpty sat on a wall, Humpty Dumpty had a great fall", 30); //Humpty Dumpty had a great fall	
to-upper-case($string)	This function will change the format of the string to all upper case
Example: to-upper-case("you can't handle the truth!"); //"YOU CAN'T HANDLE THE TRUTH"	
to-lower-case($string)	This function will chnge the format of the string to all lower case
Example: to-lower-case("SHHHH, WE'RE HUNTING WABBITS!"); //"shhhh, we're hunting wabbits!"	

Number Functions

Function defintion	Function description
percentage($number)	This will take a number which should be unitless and converts it into a precentage value. If you are dividing two numbers both of those numbers have to either be unitless or with a unit.

Example:
percentage(0.625); //62.5%
percentage(50px/100px); //50%	

round($number)	Rounds the number to the nearest whole number. This function will round up if 0.5 or above and round down if 0.4 or below. If you want more control over the direction use `ciel` or `floor` functions.

Example:
round(10.5); //11
round(10.3); //10	

ceil($number)	This function will always round up to the closest whole number.

Example:
ciel(10.25); //11
ciel(10.89); //11	

floor($number)	This function will always round down to the closest whole number.

Example:
floor(10.25); //10
floor(10.89); //10	

abs($number)	This function will return the absolute value of a number.

Example:
abs(10px); //10px
abs(-10px); //10px	

min($numbers…)	This function will return the smallest value of a series of numbers.
Example: min(0.625em, 0.75em, 1em); //0.625em	
max($numbers…)	This function will return the largest value of a series of numbers.
Example: max(0.625em, 0.75em, 1em); //1em	

List functions

Function defintion	Function description
length($list)	This function return the count of the items in a list.

Example:
length((1, 2, 3, 4, 5, 6, 7, 8, 9, 10)); //10
length((R "red") (G "green") (B "blue") (a "alpha")); //4	

nth($list, $n)	This function extracts the item in the list that is located at the given index value of $n. An error is thrown if the $n value is out of bounds of the length of the list.

Example:
nth((1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 6); //6
nth((R "red") (G "green") (B "blue") (a "alpha"), 2); //("G" "green")	

join($list1, $list2, [$separator])	This function will join two lists together. If the $seperator value is not provided and the first list is comma seperated and the second one is space seperated, the resulting list will be based on the first list, comma seperated. The values for the $seperator can be `comma` , `space` , or `auto` (which will implemented as described above).

Example:
join((1,2,3,4,5), (6 7 8 9 10)); //(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
join(0.625em, 0.75em, space); //(0.625em 0.75em)	

append($list1, $val, [$separator])	This function will allow the addition of a value to the end of a list. The rules for the $seperator follows the description mentioned in the join function.

Example:
append((1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 11); //(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
append((0.625em 0.75em), 0.825em, space); //(0.625em 0.75em 0.825em)	

zip($lists…)	This function will take mutliple lists and create a single multidimensional list with them. It is recommended that when using this function, the lists have the same length otherwise the resulting list will have the length of the smallest list.
Example: zip("red" "green" "blue", 178 18 204, 90% 50% 75%); //(("red" 178 90%), ("green" 18 50%), ("blue" 204 75%))	
index($list, $value)	This function will return the index value of the give $value within the $list. If the value is not found, it will return `null` .

Example:
index((0.625em 0.75em 0.825em), 0.75em); //2
index((0.625em 0.75em 0.825em), 1em); //null	

list-separator($list)	This function will return the seperator of the list. This value can be either `space` or `comma` . If the length of the list is less than 2 then the value returned is space.

Example:
list-separator((0.625em 0.75em 0.825em)); //space
list-separator((1, 2, 3, 4, 5, 6, 7, 8, 9, 10), 11); //comma	

Map functions

Function defintion	Function description
Example:	
map-get($map, $key)	
Example:	
map-merge($map1, $map2)	
Example:	
map-remove($map, $key)	
Example:	
map-keys($map)	
Example:	
map-values($map)	
Example:	
map-has-key($map, $key)	
Example:	
keywords($args)	
Example:	

Introspection functions

Function defintion	Function description
feature-exists($feature)	
Example:	
variable-exists($name)	
Example:	
global-variable-exists($name)	
Example:	
function-exists($name)	
Example:	
mixin-exists($name)	
Example:	
inspect($value)	
Example:	
type-of($value)	
Example:	
unit($number)	
Example:	
unitless($number)	
Example:	
comparable($number1, $number2)	
Example:	
call($name, $args…)	
Example:	

Miscallaneous functions

Function defintion	Function description
if($condition, $if-true, $if-false)	
Example:	
unique-id()	
Example:	