# **Sass Introduction**

What is Sass?

* **Sass** stands for **S**yntactically **A**wesome **S**tyle**s**heet
* Sass is an extension to CSS
* Sass is a CSS pre-processor
* Sass is completely compatible with all versions of CSS
* Sass reduces repetition of CSS and therefore saves time
* Sass was designed by Hampton Catlin and developed by Natalie Weizenbaum in 2006
* Sass is free to download and use
* Sass is a CSS pre-processor.
* Sass reduces repetition of CSS and therefore saves time.

System Requirements for Sass

* **Operating system** - Sass is platform independent
* **Browser support** - Sass works in Edge/IE (from IE 8), Firefox, Chrome, Safari, Opera
* **Programming language** - Sass is based on Ruby
* **Variables**: With SASS, you can save information in variables to be used again later. So it’s possible, for example, to centrally store a color value under a catchier variable.
* **Mathematical functions**: In SASS, you can also use mathematical operations like +, -, \*, /, or %. This allows you to influence size specifications, for example.
* **Functions**: Other functions also make it easier to work on the design. These allow you to modify color values or analyze lists, among other things.
* **Loops**: Another advantage of SASS is the ability to set up loops. These are repeated until they reach a status defined by you.
* **Case distinctions**: These have a similar function to the conditional instructions “if” and “else.” The given commands are only run if specific conditions are met.
* **Mixins**: Mixins, simply said, are templates. You can either create them yourself or simply integrate them into your own code when using a framework.
* **Indentations**: The original SASS (as opposed to SCSS) works with indentations and line breaks to structure the code. You don’t need parentheses to display nesting or semicolons to mark line ends.
* **Nesting**: CSS doesn’t allow you to work with nesting in the code. SASS, though, gives users the option to present dependencies visually to reduce redundancies and simplify the writing process.
* **Inheritances**: It’s possible to inherit properties from one selector to another. This saves some writing effort and keeps the code slimmer.
* **Partial files**: To integrate code elements into a SASS file, you can also use partials. These are files that only have to contain a few lines of CSS and are imported into a SASS file by command.

Sass Variables

Variables are a way to store information that you can re-use later.

With Sass, you can store information in variables, like:

* strings
* numbers
* colors
* booleans
* lists
* nulls

Sass uses the $ symbol, followed by a name, to declare variables:

Sass Variable Syntax:

$*variablename*: *value*;

The following example declares 4 variables named myFont, myColor, myFontSize, and myWidth. After the variables are declared, you can use the variables wherever you want:

SCSS Syntax:

$myFont: Helvetica, sans-serif;  
$myColor: red;  
$myFontSize: 18px;  
$myWidth: 680px;  
  
body {  
  font-family: $myFont;  
  font-size: $myFontSize;  
  color: $myColor;  
}  
  
#container {  
  width: $myWidth;  
}

## Using Sass !global

The default behavior for variable scope can be overridden by using the !global switch.

!global indicates that a variable is global, which means that it is accessible on all levels.

Look at the following example (same as above; but with !global added):

SCSS Syntax:

$myColor: red;  
  
h1 {  
  $myColor: green !global;  
  color: $myColor;  
}  
  
p {  
  color: $myColor;  
}

# **Sass Nested Rules and Properties**

## Sass Nested Rules

Sass lets you nest CSS selectors in the same way as HTML.

Look at an example of some Sass code for a site's navigation:

### **Example**

SCSS Syntax:

nav {  
  ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
  }  
  li {  
    display: inline-block;  
  }  
  a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
  }  
}

Notice that in Sass, the ul, li, and a selectors are nested inside the nav selector.

While in CSS, the rules are defined one by one (not nested):

CSS Syntax:

nav ul {  
  margin: 0;  
  padding: 0;  
  list-style: none;  
}  
nav li {  
  display: inline-block;  
}  
nav a {  
  display: block;  
  padding: 6px 12px;  
  text-decoration: none;  
}

## Sass Nested Properties

Many CSS properties have the same prefix, like font-family, font-size and font-weight or text-align, text-transform and text-overflow.

With Sass you can write them as nested properties:

### **Example**

SCSS Syntax:

font: {  
  family: Helvetica, sans-serif;  
  size: 18px;  
  weight: bold;  
}  
  
text: {  
  align: center;  
  transform: lowercase;  
  overflow: hidden;  
}

# **Sass @import and Partials**

Sass keeps the CSS code DRY (Don't Repeat Yourself). One way to write DRY code is to keep related code in separate files.

You can create small files with CSS snippets to include in other Sass files. Examples of such files can be: reset file, variables, colors, fonts, font-sizes, etc.

Sass Importing Files

Just like CSS, Sass also supports the @import directive.

The @import directive allows you to include the content of one file in another.

The CSS @import directive has a major drawback due to performance issues; it creates an extra HTTP request each time you call it. However, the Sass @import directive includes the file in the CSS; so no extra HTTP call is required at runtime!

Sass Import Syntax:

@import *filename*;

Sass automatically assumes that you mean a .sass or .scss file. You can also import CSS files. The @import directive imports the file and any variables or mixins defined in the imported file can then be used in the main file.

You can import as many files as you need in the main file:

### **Example**

@import "variables";  
@import "colors";  
@import "reset";

Let's look at an example: Let's assume we have a reset file called "reset.scss", that looks like this:

### **Example**

SCSS Syntax (reset.scss):

html,  
body,  
ul,  
ol {  
  margin: 0;  
  padding: 0;  
}

and now we want to import the "reset.scss" file into another file called "standard.scss".

Here is how we do it: It is normal to add the @import directive at the top of a file; this way its content will have a global scope:

SCSS Syntax (standard.scss):

@import "reset";  
  
body {  
  font-family: Helvetica, sans-serif;  
  font-size: 18px;  
  color: red;  
}

So, when the "standard.css" file is transpiled, the CSS will look like this:

CSS output:

html, body, ul, ol {  
  margin: 0;  
  padding: 0;  
}  
  
body {  
  font-family: Helvetica, sans-serif;  
  font-size: 18px;  
  color: red;  
}

# **Sass @mixin and @include**

## Sass Mixins

The @mixin directive lets you create CSS code that is to be reused throughout the website.

The @include directive is created to let you use (include) the mixin.

Defining a Mixin

A mixin is defined with the @mixin directive.

Sass @mixin Syntax:

@mixin *name*{  
  *property*: *value*;  
  *property*: *value*;  
  ...  
}

The following example creates a mixin named "important-text":

SCSS Syntax:

@mixin important-text {  
  color: red;  
  font-size: 25px;  
  font-weight: bold;  
  border: 1px solid blue;  
}

**Tip:** A tip on hyphens and underscore in Sass: Hyphens and underscores are considered to be the same. This means that @mixin important-text { } and @mixin important\_text { } are considered as the same mixin!

Using a Mixin

The @include directive is used to include a mixin.

Sass @include mixin Syntax:

*selector*{  
  @include *mixin-name*;}

So, to include the important-text mixin created above:

SCSS Syntax:

.danger {  
  @include important-text;  
  background-color: green;  
}

The Sass transpiler will convert the above to normal CSS:

CSS output:

.danger {  
  color: red;  
  font-size: 25px;  
  font-weight: bold;  
  border: 1px solid blue;  
  background-color: green;  
}

A mixin can also include other mixins:

SCSS Syntax:

@mixin special-text {  
  @include important-text;  
  @include link;  
  @include special-border;  
}

Passing Variables to a Mixin

Mixins accept arguments. This way you can pass variables to a mixin.

Here is how to define a mixin with arguments:

SCSS Syntax:

/\* Define mixin with two arguments \*/  
@mixin bordered($color, $width) {  
  border: $width solid $color;  
}  
  
.myArticle {  
  @include bordered(blue, 1px);  // Call mixin with two values  
}  
  
.myNotes {  
  @include bordered(red, 2px); // Call mixin with two values  
}

Notice that the arguments are set as variables and then used as the values (color and width) of the border property.

After compilation, the CSS will look like this:

CSS Output:

.myArticle {  
  border: 1px solid blue;  
}  
  
.myNotes {  
  border: 2px solid red;  
}

# **Sass @extend and Inheritance**

Sass @extend Directive

The @extend directive lets you share a set of CSS properties from one selector to another.

The @extend directive is useful if you have almost identically styled elements that only differ in some small details.

The following Sass example first creates a basic style for buttons (this style will be used for most buttons). Then, we create one style for a "Report" button and one style for a "Submit" button. Both "Report" and "Submit" button inherit all the CSS properties from the .button-basic class, through the @extend directive. In addition, they have their own colors defined:

SCSS Syntax:

.button-basic  {  
  border: none;  
  padding: 15px 30px;  
  text-align: center;  
  font-size: 16px;  
  cursor: pointer;  
}  
  
.button-report  {  
  @extend .button-basic;  
  background-color: red;  
}  
  
.button-submit  {  
  @extend .button-basic;  
  background-color: green;  
  color: white;  
}

After compilation, the CSS will look like this:

CSS Output:

.button-basic, .button-report, .button-submit {  
  border: none;  
  padding: 15px 30px;  
  text-align: center;  
  font-size: 16px;  
  cursor: pointer;  
}  
  
.button-report  {  
  background-color: red;  
}  
  
.button-submit  {  
  background-color: green;  
  color: white;  
}

By using the @extend directive, you do not need to specify several classes for an element in your HTML code, like this: <button class="button-basic button-report">Report this</button>. You just need to specify .button-report to get both sets of styles.

The @extend directive helps keep your Sass code very DRY.

# **Sass String Functions**

Sass String Functions

The string functions are used to manipulate and get information about strings.

Sass strings are 1-based. The first character in a string is at index 1, not 0.

The following table lists all string functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| quote(*string*) | Adds quotes to *string*, and returns the result.  **Example:** quote(Hello world!) Result: "Hello world!" |
| str-index(*string*,*substring*) | Returns the index of the first occurrence of the *substring* within *string*.  **Example:** str-index("Hello world!", "H") Result: 1 |
| str-insert(*string*, *insert*, *index*) | Returns *string* with *insert* inserted at the specified *index* position.  **Example:** str-insert("Hello world!", " wonderful", 6) Result: "Hello wonderful world!" |
| str-length(*string*) | Returns the length of *string* (in characters).  **Example:** str-length("Hello world!") Result: 12 |
| str-slice(*string*, *start*, *end*) | Extracts characters from *string*; start at *start* and end at *end*, and returns the slice.  **Example:** str-slice("Hello world!", 2, 5) Result: "ello" |
| to-lower-case(*string*) | Returns a copy of *string* converted to lower case.  **Example:** to-lower-case("Hello World!") Result: "hello world!" |
| to-upper-case(*string*) | Returns a copy of *string* converted to upper case.  **Example:** to-upper-case("Hello World!") Result: "HELLO WORLD!" |
| unique-id() | Returns a unique randomly generated unquoted string (guaranteed to be unique within the current sass session).  **Example:** unique-id() Result: tyghefnsv |
| unquote(*string*) | Removes quotes around *string* (if any), and returns the result.  **Example:** unquote("Hello world!") Result: Hello world! |

Sass Numeric Functions

The numeric functions are used to manipulate numeric values.

The following table lists all numeric functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| abs(*number*) | Returns the absolute value of *number*.  **Example:** abs(15) Result: 15 abs(-15) Result: 15 |
| ceil(*number*) | Rounds *number* up to the nearest  integer.  **Example:** ceil(15.20) Result: 16 |
| comparable(*num1*, *num2*) | Returns whether *num1* and *num2* are comparable.  **Example:** comparable(15px, 10px) Result: true comparable(20mm, 1cm) Result: true comparable(35px, 2em) Result: false |
| floor(*number*) | Rounds *number* down to the nearest integer.  **Example:** floor(15.80) Result: 15 |
| max(*number...*) | Returns the highest value of several numbers.  **Example:** max(5, 7, 9, 0, -3, -7) Result: 9 |
| min(*number...*) | Returns the lowest value of several numbers.  **Example:** min(5, 7, 9, 0, -3, -7) Result: -7 |
| percentage(*number*) | Converts *number* to a percentage (multiplies the number with 100).  **Example:** percentage(1.2) Result: 120 |
| random() | Returns a random number between 0 and 1.  **Example:** random() Result: 0.45673 |
| random(*number*) | Returns a random integer between 1 and *number*.  **Example:** random(6) Result: 4 |
| round(*number*) | Rounds *number* to the nearest integer.  **Example:** round(15.20) Result: 15 round(15.80) Result: 16 |

# **Sass List Functions**

Sass List Functions

The list functions are used to access values in a list, combine lists, and add items to lists.

Sass lists are immutable (they cannot change). So, the list functions that return a list, will return a new list, and not change the original list.

Sass lists are 1-based. The first list item in a list is at index 1, not 0.

The following table lists all list functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| append(*list*, *value*, [*separator*]) | Adds a single *value* to the end of the list. *separator* can be auto, comma, or space. auto is default.  **Example:** append((a b c), d) Result: a b c d append((a b c), (d), comma) Result: a, b, c, d |
| index(*list*, *value*) | Returns the index position for the *value* in list.  **Example:** index(a b c, b) Result: 2 index(a b c, f) Result: null |
| is-bracketed(*list*) | Checks whether the list has square brackets.  **Example:** is-bracketed([a b c]) Result: true is-bracketed(a b c) Result: false |
| join(*list1*, *list2*, [*separator, bracketed*]) | Appends *list2* to the end of *list1*. *separator* can be auto, comma, or space. auto is default (will use the separator in the first list). *bracketed* can be auto, true, or false. auto is default.  **Example:** join(a b c, d e f) Result: a b c d e f join((a b c), (d e f), comma) Result: a, b, c, d, e, f join(a b c, d e f, $bracketed: true) Result: [a b c d e f] |
| length(*list*) | Returns the length of the list.  **Example:** length(a b c) Result: 3 |
| list-separator(*list*) | Returns the list separator used, as a string. Can be either space or comma.  **Example:** list-separator(a b c) Result: "space" list-separator(a, b, c) Result: "comma" |
| nth(*list*, *n*) | Returns the *n*th element in the list.  **Example:** nth(a b c, 3) Result: c |
| set-nth(*list*, *n*, *value*) | Sets the *n*th list element to the *value* specified.  **Example:** set-nth(a b c, 2, x) Result: a x c |
| zip(*lists*) | Combines lists into a single multidimensional list.  **Example:** zip(1px 2px 3px, solid dashed dotted, red green blue) Result: 1px solid red, 2px dashed green, 3px dotted blue |

# **Sass Map Functions**

Sass Map Functions

In Sass, the map data type represents one or more key/value pairs.

**Tip:** It is also possible to use the [List functions](https://www.w3schools.com/sass/sass_functions_list.php) from the previous page, with maps. Then the map will be treated as a list with two elements.

Sass maps are immutable (they cannot change). So, the map functions that return a map, will return a new map, and not change the original map.

The following table lists all map functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| map-get(*map*, *key*) | Returns the value for the specified *key*in the map.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-get($font-sizes, "small") Result: 12px |
| map-has-key(*map*, *key*) | Checks whether *map* has the specified *key*. Returns true or false.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-has-key($font-sizes, "big") Result: false |
| map-keys(*map*) | Returns a list of all keys in *map*.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-keys($font-sizes) Result: "small", "normal, "large" |
| map-merge(*map1*, *map2*) | Appends *map2* to the end of *map1*.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) $font-sizes2: ("x-large": 30px, "xx-large": 36px) map-merge($font-sizes, $font-sizes2) Result: "small": 12px, "normal": 18px, "large": 24px, "x-large": 30px, "xx-large": 36px |
| map-remove(*map*, *keys...*) | Removes the specified keys from *map*.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-remove($font-sizes, "small") Result: ("normal": 18px, "large": 24px) map-remove($font-sizes, "small", "large") Result: ("normal": 18px) |
| map-values(*map*) | Returns a list of all values in *map*.  **Example:** $font-sizes: ("small": 12px, "normal": 18px, "large": 24px) map-values($font-sizes) Result: 12px, 18px, 24px |

# **Sass Selector Functions**

Sass Selector Functions

The selector functions are used to check and manipulate selectors.

The following table lists all selector functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| is-superselector(*super*, *sub*) | Checks whether the *super* selector matches all the elements that *sub* matches.  **Example:** is-superselector("div", "div.myInput") Result: true is-superselector("div.myInput", "div") Result: false is-superselector("div", "div") Result: true |
| selector-append(*selectors*) | Appends the second (and third/fourth etc.) selector to the first selector.  **Example:** selector-append("div", ".myInput") Result: div.myInput selector-append(".warning", "\_\_a") Result: .warning\_\_a |
| selector-extend(*selector*, *extendee*, *extender*) |  |
| selector-nest(*selectors*) | Returns a new selector containing a nested list of CSS selectors based on the list provided.  **Example:** selector-nest("ul", "li") Result: ul li selector-nest(".warning", "alert", "div") Result: .warning div, alert div |
| selector-parse(*selector*) | Returns a list of strings contained in *selector* using the same format as the parent selector.  **Example:** selector-parse("h1 .myInput .warning") Result: ('h1' '.myInput' '.warning') |
| selector-replace(*selector*, *original*, *replacement*) | Returns a new selector with the selectors specified in *replacement* in place of selectors specified in *original*.  **Example:** selector-replace("p.warning", "p", "div") Result: div.warning |
| selector-unify(*selector1*, *selector2*) | Returns a new selector that matches only elements matched by both *selector1* and *selector2*.  **Example:** selector-unify("myInput", ".disabled") Result: myInput.disabled selector-unify("p", "h1") Result: null |
| simple-selectors(*selectors*) | Returns a list of the individual selectors in *selectors*.  **Example:** simple-selectors("div.myInput") Result: div, .myInput simple-selectors("div.myInput:before") Result: div, .myInput, :before |

# **Sass Introspection Functions**

Sass Introspection Functions

The introspection functions are rarely used when building a stylesheet. However, they are valuable if something does not work properly - to figure out what's going on: like debugging functions.

The following table lists all introspection functions in Sass:

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| call(*function*, *arguments*...) | Calls a function with arguments, and returns the result. |
| content-exists() | Checks whether the current mixin was passed a @content block. |
| feature-exists(*feature*) | Checks whether *feature* is supported by the current Sass implementation.  **Example:** feature-exists("at-error"); Result: true |
| function-exists(*functionname*) | Checks whether the specified function exists.  **Example:** function-exists("nonsense") Result: false |
| get-function(*functionname*, css: false) | Returns the specified function. If css is true, it returns a plain CSS function instead. |
| global-variable-exists(*variablename*) | Checks whether the specified global variable exists.  **Example:** variable-exists(a) Result: true |
| inspect(*value*) | Returns a string representation of *value*. |
| mixin-exists(*mixinname*) | Checks whether the specified mixin exists.  **Example:** mixin-exists("important-text") Result: true |
| type-of(*value*) | Returns the type of *value*. Can be number, string, color, list, map, bool, null, function, arglist.  **Example:** type-of(15px) Result: number type-of(#ff0000) Result: color |
| unit(*number*) | Returns the unit associated with a number.  **Example:** unit(15px) Result: px |
| unitless(*number*) | Checks whether the specified number has a unit associated with it.  **Example:** unitless(15px) Result: false unitless(15) Result: true |
| variable-exists(*variablename*) | Checks whether the specified variable exists in the current scope.  **Example:** variable-exists(b) Result: true |

# **Sass Color Functions**

## Sass Color Functions

We have divided the color functions in Sass into three parts: Set color functions, Get color functions, and Manipulate color functions:

### **Sass Set Color Functions**

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| rgb(red, green, blue) | Sets a color using the Red-Green-Blue (RGB) model. An RGB color value is specified with: rgb(red, green, blue). Each parameter defines the intensity of that color and can be an integer between 0 and 255, or a percentage value (from 0% to 100%).  **Example:** rgb(0, 0, 255); // rendered as blue because the blue parameter is set to its highest value (255) and the others are set to 0 |
| rgba(red, green, blue, alpha) | Sets a color using the Red-Green-Blue-Alpha (RGBA) model. RGBA color values are an extension of RGB color values with an alpha channel - which specifies the opacity of the color. The alpha parameter is a number between 0.0 (fully transparent) and 1.0 (fully opaque).  **Example:** rgba(0, 0, 255, 0.3); // rendered as blue with opacity |
| hsl(hue, saturation, lightness) | Sets a color using the Hue-Saturation-Lightness (HSL) model - and represents a cylindrical-coordinate representation of colors. Hue is a degree on the color wheel (from 0 to 360) - 0 or 360 is red, 120 is green, 240 is blue. Saturation is a percentage value; 0% means a shade of gray and 100% is the full color. Lightness is also a percentage; 0% is black, 100% is white.  **Example:** hsl(120, 100%, 50%); // green hsl(120, 100%, 75%); // light green hsl(120, 100%, 25%); // dark green hsl(120, 60%, 70%); // pastel green |
| hsla(hue, saturation, lightness, alpha) | Sets a color using the Hue-Saturation-Lightness-Alpha (HSLA) model. HSLA color values are an extension of HSL color values with an alpha channel - which specifies the opacity of the color. The alpha parameter is a number between 0.0 (fully transparent) and 1.0 (fully opaque).  **Example:** hsl(120, 100%, 50%, 0.3); // green with opacity hsl(120, 100%, 75%, 0.3); // light green with opacity |
| grayscale(color) | Sets a gray color with the same lightness as color.  **Example:** grayscale(#7fffd4); Result: #c6c6c6 |
| complement(color) | Sets a color that is the complementary color of color.  **Example:** complement(#7fffd4); Result: #ff7faa |
| invert(color, weight) | Sets a color that is the inverse or negative color of color. The weight parameter is optional and must be a number between 0% and 100%. Default is 100%.  **Example:** invert(white); Result: black |

### **Sass Get Color Functions**

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| red(color) | Returns the red value of color as a number between 0 and 255.  **Example:** red(#7fffd4); Result: 127 red(red); Result: 255 |
| green(color) | Returns the green value of color as a number between 0 and 255.  **Example:** green(#7fffd4); Result: 255 green(blue); Result: 0 |
| blue(color) | Returns the blue value of color as a number between 0 and 255.  **Example:** blue(#7fffd4); Result: 212 blue(blue); Result: 255 |
| hue(color) | Returns the hue of color as a number between 0deg and 255deg.  **Example:** hue(#7fffd4); Result: 160deg |
| saturation(color) | Returns the HSL saturation of color as a number between 0% and 100%.  **Example:** saturation(#7fffd4); Result: 100% |
| lightness(color) | Returns the HSL lightness of color as a number between 0% and 100%.  **Example:** lightness(#7fffd4); Result: 74.9% |
| alpha(color) | Returns the alpha channel of color as a number between 0 and 1.  **Example:** alpha(#7fffd4); Result: 1 |
| opacity(color) | Returns the alpha channel of color as a number between 0 and 1.  **Example:** opacity(rgba(127, 255, 212, 0.5)); Result: 0.5 |

### **Sass Manipulate Color Functions**

|  |  |
| --- | --- |
| **Function** | **Description & Example** |
| mix(color1, color2, weight) | Creates a color that is a mix of color1 and color2. The weight parameter must be between 0% and 100%. A larger weight means that more of color1 should be used. A smaller weight means that more of color2 should be used. Default is 50%. |
| adjust-hue(color, degrees) | Adjusts the color's hue with a degree from -360deg to 360deg.  **Example:** adjust-hue(#7fffd4, 80deg); Result: #8080ff |
| adjust-color(color, red, green, blue, hue, saturation, lightness, alpha) | Adjusts one or more parameters by the specified amount. This function adds or subtracts the specified amount to/from the existing color value.  **Example:** adjust-color(#7fffd4, blue: 25); Result: |
| change-color(color, red, green, blue, hue, saturation, lightness, alpha) | Sets one or more parameters of a color to new values.  **Example:** change-color(#7fffd4, red: 255); Result: #ffffd4 |
| scale-color(color, red, green, blue,  saturation, lightness, alpha) | Scales one or more parameters of color. |
| rgba(color, alpha) | Creates a new color of color with the given alpha channel.  **Example:** rgba(#7fffd4, 30%); Result: rgba(127, 255, 212, 0.3) |
| lighten(color, amount) | Creates a lighter color of color with an amount between 0% and 100%. The amount parameter increases the HSL lightness by that percent. |
| darken(color, amount) | Creates a darker color of color with an amount between 0% and 100%. The amount parameter decreases the HSL lightness by that percent. |
| saturate(color, amount) | Creates a more saturated color of color with an amount between 0% and 100%. The amount parameter increases the HSL saturation by that percent. |
| desaturate(color, amount) | Creates a less saturated color of color with an amount between 0% and 100%. The amount parameter decreases the HSL saturation by that percent. |
| opacify(color, amount) | Creates a more opaque color of color with an amount between 0 and 1. The amount parameter increases the alpha channel by that amount. |
| fade-in(color, amount) | Creates a more opaque color of color with an amount between 0 and 1. The amount parameter increases the alpha channel by that amount. |
| transparentize(color, amount) | Creates a more transparent color of color with an amount between 0 and 1. The amount parameter decreases the alpha channel by that amount. |
| fade-out(color, amount) | Creates a more transparent color of color with an amount between 0 and 1. The amount parameter decreases the alpha channel by that amount. |