**CSS Extensions**

**Nested Rules**

Sass allows CSS rules to be nested within one another. The inner rule then only applies within the outer rule’s selector. For example:

#main p {

color: #00ff00;

width: 97%;

.redbox {

background-color: #ff0000;

color: #000000;

}

}

is compiled to:

#main p {

color: #00ff00;

width: 97%; }

#main p .redbox {

background-color: #ff0000;

color: #000000; }

This helps avoid repetition of parent selectors, and makes complex CSS layouts with lots of nested selectors much simpler. For example:

#main {

width: 97%;

p, div {

font-size: 2em;

a { font-weight: bold; }

}

pre { font-size: 3em; }

}

is compiled to:

#main {

width: 97%; }

#main p, #main div {

font-size: 2em; }

#main p a, #main div a {

font-weight: bold; }

#main pre {

font-size: 3em; }

**Referencing Parent Selectors: &**

Sometimes it’s useful to use a nested rule’s parent selector in other ways than the default. For instance, you might want to have special styles for when that selector is hovered over or for when the body element has a certain class. In these cases, you can explicitly specify where the parent selector should be inserted using the & character. For example:

a {

font-weight: bold;

text-decoration: none;

&:hover { text-decoration: underline; }

body.firefox & { font-weight: normal; }

}

is compiled to:

a {

font-weight: bold;

text-decoration: none; }

a:hover {

text-decoration: underline; }

body.firefox a {

font-weight: normal; }

& will be replaced with the parent selector as it appears in the CSS. This means that if you have a deeply nested rule, the parent selector will be fully resolved before the & is replaced. For example:

#main {

color: black;

a {

font-weight: bold;

&:hover { color: red; }

}

}

is compiled to:

#main {

color: black; }

#main a {

font-weight: bold; }

#main a:hover {

color: red; }

**Nested Properties**

CSS has quite a few properties that are in “namespaces;” for instance, font-family, font-size, and font-weight are all in the font namespace. In CSS, if you want to set a bunch of properties in the same namespace, you have to type it out each time. Sass provides a shortcut for this: just write the namespace one, then nest each of the sub-properties within it. For example:

.funky {

font: {

family: fantasy;

size: 30em;

weight: bold;

}

}

is compiled to:

.funky {

font-family: fantasy;

font-size: 30em;

font-weight: bold; }

The property namespace itself can also have a value. For example:

.funky {

font: 2px/3px {

family: fantasy;

size: 30em;

weight: bold;

}

}

is compiled to:

.funky {

font: 2px/3px;

font-family: fantasy;

font-size: 30em;

font-weight: bold; }

**Comments: /\* \*/ and //**

Sass supports standard multiline CSS comments with /\* \*/, as well as single-line comments with //. The multiline comments are preserved in the CSS output where possible, while the single-line comments are removed. For example:

/\* This comment is

\* several lines long.

\* since it uses the CSS comment syntax,

\* it will appear in the CSS output. \*/

body { color: black; }

// These comments are only one line long each.

// They won't appear in the CSS output,

// since they use the single-line comment syntax.

a { color: green; }

is compiled to:

/\* This comment is

\* several lines long.

\* since it uses the CSS comment syntax,

\* it will appear in the CSS output. \*/

body {

color: black; }

a {

color: green; }

**SassScript**

In addition to the plain CSS property syntax, Sass supports a small set of extensions called SassScript. SassScript allows properties to use variables, arithmetic, and extra functions. SassScript can be used in any property value.

SassScript can also be used to generate selectors and property names, which is useful when writing [mixins](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixins). This is done via [interpolation](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#interpolation_).

**Interactive Shell**

You can easily experiment with SassScript using the interactive shell. To launch the shell run the sass command-line with the -i option. At the prompt, enter any legal SassScript expression to have it evaluated and the result printed out for you:

$ sass -i

>> "Hello, Sassy World!"

"Hello, Sassy World!"

>> 1px + 1px + 1px

3px

>> #777 + #777

#eeeeee

>> #777 + #888

white

**Variables: $**

The most straightforward way to use SassScript is to use variables. Variables begin with dollar signs, and are set like CSS properties:

$width: 5em;

You can then refer to them in properties:

#main {

width: $width;

}

Variables are only available within the level of nested selectors where they’re defined. If they’re defined outside of any nested selectors, they’re available everywhere.

Variables used to use the prefix character !; this still works, but it’s deprecated and prints a warning. $ is the recommended syntax.

Variables also used to be defined with = rather than :; this still works, but it’s deprecated and prints a warning. : is the recommended syntax.

**Data Types**

SassScript supports four main data types:

* numbers (e.g. 1.2, 13, 10px)
* strings of text, with and without quotes (e.g. "foo", 'bar', baz)
* colors (e.g. blue, #04a3f9, rgba(255, 0, 0, 0.5))
* booleans (e.g. true, false)

SassScript also supports all other types of CSS property value, such as Unicode ranges and !important declarations. However, it has no special handling for these types. They’re treated just like unquoted strings.

**Strings**

CSS specifies two kinds of strings: those with quotes, such as "Lucida Grande" or 'http://sass-lang.com', and those without quotes, such as sans-serif or bold. SassScript recognizes both kinds, and in general if one kind of string is used in the Sass document, that kind of string will be used in the resulting CSS.

There is one exception to this, though: when using [#{} interpolation](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#interpolation_), quoted strings are unquoted. This makes it easier to use e.g. selector names in [mixins](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixins). For example:

@mixin firefox-message($selector) {

body.firefox #{$selector}:before {

content: "Hi, Firefox users!"; } }

@include firefox-message(".header");

is compiled to:

body.firefox .header:before {

content: "Hi, Firefox users!"; }

It’s also worth noting that when using the [deprecated = property syntax](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#sassscript), all strings are interpreted as unquoted, regardless of whether or not they’re written with quotes.

**Operations**

All types support equality operations (== and !=). In addition, each type has its own operations that it has special support for.

**Number Operations**

SassScript supports the standard arithmetic operations on numbers (+, -, \*, /, %), and will automatically convert between units if it can:

p {

width: 1in + 8pt;

}

is compiled to:

p {

width: 1.111in; }

Relational operators (<, >, <=, >=) are also supported for numbers, and equality operators (==, !=) are supported for all types.

**Division and /**

CSS allows / to appear in property values as a way of separating numbers. Since SassScript is an extension of the CSS property syntax, it must support this, while also allowing / to be used for division. This means that by default, if two numbers are separated by / in SassScript, then they will appear that way in the resulting CSS.

However, there are three situations where the / will be interpreted as division. These cover the vast majority of cases where division is actually used. They are:

1. If the value, or any part of it, is stored in a variable.
2. If the value is surrounded by parentheses.
3. If the value is used as part of another arithmetic expression.

For example:

p {

font: 10px/8px; // Plain CSS, no division

$width: 1000px;

width: $width/2; // Uses a variable, does division

height: (500px/2); // Uses parentheses, does division

margin-left: 5px + 8px/2px; // Uses +, does division

}

is compiled to:

p {

font: 10px/8px;

width: 500px;

height: 250px;

margin-left: 9px; }

If you want to use variables along with a plain CSS /, you can use #{} to insert them. For example:

p {

$font-size: 12px;

$line-height: 30px;

font: #{$font-size}/#{$line-height};

}

is compiled to:

p {

font: 12px/30px;

}

**Color Operations**

All arithmetic operations are supported for color values, where they work piecewise. This means that the operation is performed on the red, green, and blue components in turn. For example:

p {

color: #010203 + #040506;

}

computes 01 + 04 = 05, 02 + 05 = 07, and 03 + 06 = 09, and is compiled to:

p {

color: #050709; }

Often it’s more useful to use [color functions](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html) than to try to use color arithmetic to achieve the same effect.

Arithmetic operations also work between numbers and colors, also piecewise. For example:

p {

color: #010203 \* 2;

}

computes 01 \* 2 = 02, 02 \* 2 = 04, and 03 \* 2 = 06, and is compiled to:

p {

color: #020406; }

Note that colors with an alpha channel (those created with the [rgba](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html#rgba-instance_method) or [hsla](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html#hsla-instance_method) functions) must have the same alpha value in order for color arithmetic to be done with them. The arithmetic doesn’t affect the alpha value. For example:

p {

color: rgba(255, 0, 0, 0.75) + rgba(0, 255, 0, 0.75);

}

is compiled to:

p {

color: rgba(255, 255, 0, 0.75); }

The alpha channel of a color can be adjusted using the [opacify](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html#opacify-instance_method) and [transparentize](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html#transparentize-instance_method) functions. For example:

$translucent-red: rgba(255, 0, 0, 0.5);

p {

color: opacify($translucent-red, 0.8);

background-color: transparentize($translucent-red, 50%);

}

is compiled to:

p {

color: rgba(255, 0, 0, 0.9);

background-color: rgba(255, 0, 0, 0.25); }

**String Operations**

The + operation can be used to concatenate strings:

p {

cursor: e + -resize;

}

is compiled to:

p {

cursor: e-resize; }

Note that if a quoted string is added to an unquoted string (that is, the quoted string is to the left of the +), the result is a quoted string. Likewise, if an unquoted string is added to a quoted string (the unquoted string is to the left of the +), the result is an unquoted string. For example:

p:before {

content: "Foo " + Bar;

font-family: sans- + "serif"; }

is compiled to:

p:before {

content: "Foo Bar";

font-family: sans-serif; }

By default, if two values are placed next to one another, they are concatenated with a space:

p {

margin: 3px + 4px auto;

}

is compiled to:

p {

margin: 7px auto; }

Within a string of text, #{} style interpolation can be used to place dynamic values within the string:

p:before {

content: "I ate #{5 + 10} pies!"; }

is compiled to:

p:before {

content: "I ate 15 pies!"; }

**Boolean Operations**

SassScript supports and, or, and not operators for boolean values.

**Parentheses**

Parentheses can be used to affect the order of operations:

p {

width: 1em + (2em \* 3);

}

is compiled to:

p {

width: 7em; }

**Functions**

SassScript defines some useful functions that are called using the normal CSS function syntax:

p {

color: hsl(0, 100%, 0.5);

}

is compiled to:

p {

color: #ff0000; }

See [Sass::Script::Functions](http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html) for a full listing of Sass functions, as well as instructions on defining your own in Ruby.

**Interpolation: #{}**

You can also use SassScript variables in selectors and property names using #{} interpolation syntax:

$name: foo;

$attr: border;

p.#{$name} { #{$attr}-color: blue }

is compiled to:

p.foo {

border-color: blue; }

It’s also possible to use #{} to put SassScript into property values. In most cases this isn’t any better than using a variable, but using #{} does mean that any operations near it will be treated as plain CSS. For example:

p {

$font-size: 12px;

$line-height: 30px;

font: #{$font-size}/#{$line-height};

}

is compiled to:

p {

font: 12px/30px;

}

**Variable Defaults: !default**

You can assign to variables if they aren’t already assigned by adding the !default flag to the end of the value. This means that if the variable has already been assigned to, it won’t be re-assigned, but if it doesn’t have a value yet, it will be given one.

For example:

$content: "First content";

$content: "Second content?" !default;

$new\_content: "First time reference" !default;

#main {

content: $content;

new-content: $new\_content;

}

is compiled to:

#main {

content: "First content";

new-content: "First time reference"; }

**@-Rules and Directives**

Sass supports all CSS3 @-rules, as well as some additional Sass-specific ones known as “directives.” These have various effects in Sass, detailed below. See also [control directives](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#control-directives) and [mixin directives](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixins).

**@import**

Sass extends the CSS @import rule to allow it to import SCSS and Sass files. All imported SCSS and Sass files will be merged together into a single CSS output file. In addition, any variables or [mixins](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixins) defined in imported files can be used in the main file.

Sass looks for other Sass files in the current directory, and the Sass file directory under Rack, Rails, or Merb. Additional search directories may be specified using the [:load\_paths](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#load_paths-option) option, or the --load-path option on the command line.

@import takes a filename to import. By default, it looks for a Sass file to import directly, but there are a few circumstances under which it will compile to a CSS @import rule:

* If the file’s extension is .css.
* If the filename begins with http://.
* If the filename is a url().
* If the @import has any media queries.

If none of the above conditions are met and the extension is .scss or .sass, then the named Sass or SCSS file will be imported. If there is no extension, Sass will try to find a file with that name and the .scss or .sass extension and import it.

For example,

@import "foo.scss";

or

@import "foo";

would both import the file foo.scss, whereas

@import "foo.css";

@import "foo" screen;

@import "http://foo.com/bar";

@import url(foo);

would all compile to

@import "foo.css";

@import "foo" screen;

@import "http://foo.com/bar";

@import url(foo);

It’s also possible to import multiple files in one @import. For example:

@import "rounded-corners", "text-shadow";

would import both the rounded-corners and the text-shadow files.

**Partials**

If you have a SCSS or Sass file that you want to import but don’t want to compile to a CSS file, you can add an underscore to the beginning of the filename. This will tell Sass not to compile it to a normal CSS file. You can then import these files without using the underscore.

For example, you might have \_colors.scss. Then no \_colors.css file would be created, and you can do

@import "colors";

and \_colors.scss would be imported.

**@extend**

There are often cases when designing a page when one class should have all the styles of another class, as well as its own specific styles. The most common way of handling this is to use both the more general class and the more specific class in the HTML. For example, suppose we have a design for a normal error and also for a serious error. We might write our markup like so:

<div class="error seriousError">

Oh no! You've been hacked!

</div>

And our styles like so:

.error {

border: 1px #f00;

background-color: #fdd;

}

.seriousError {

border-width: 3px;

}

Unfortunately, this means that we have to always remember to use .error with .seriousError. This is a maintenance burden, leads to tricky bugs, and can bring non-semantic style concerns into the markup.

The @extend directive avoids these problems by telling Sass that one selector should inherit the styles of another selector. For example:

.error {

border: 1px #f00;

background-color: #fdd;

}

.seriousError {

@extend .error;

border-width: 3px;

}

This means that all styles defined for .error are also applied to .seriousError, in addition to the styles specific to .seriousError. In effect, everything with class .seriousError also has class .error.

Other rules that use .error will work for .seriousError as well. For example, if we have special styles for errors caused by hackers:

.error.intrusion {

background-image: url("/image/hacked.png");

}

Then <div class="seriousError intrusion"> will have the hacked.png background image as well.

**How it Works**

@extend works by inserting the extending selector (e.g. .seriousError) anywhere in the stylesheet that the extended selector (.e.g .error) appears. Thus the example above:

.error {

border: 1px #f00;

background-color: #fdd;

}

.error.intrusion {

background-image: url("/image/hacked.png");

}

.seriousError {

@extend .error;

border-width: 3px;

}

is compiled to:

.error, .seriousError {

border: 1px #f00;

background-color: #fdd; }

.error.intrusion, .seriousError.intrusion {

background-image: url("/image/hacked.png"); }

.seriousError {

border-width: 3px; }

When merging selectors, @extend is smart enough to avoid unnecessary duplication, so something like .seriousError.seriousError gets translated to .seriousError. In addition, it won’t produce selectors that can’t match anything, like #main#footer.

**Extending Complex Selectors**

Class selectors aren’t the only things that can be extended. It’s possible to extend any selector involving only a single element, such as .special.cool, a:hover, or a.user[href^="http://"]. For example:

.hoverlink {@extend a:hover}

Just like with classes, this means that all styles defined for a:hover are also applied to .hoverlink. For example:

.hoverlink {@extend a:hover}

a:hover {text-decoration: underline}

is compiled to:

a:hover, .hoverlink {text-decoration: underline}

Just like with .error.intrusion above, any rule that uses a:hover will also work for .hoverlink, even if they have other selectors as well. For example:

.hoverlink {@extend a:hover}

.comment a.user:hover {font-weight: bold}

is compiled to:

.comment a.user:hover, .comment .hoverlink.user {font-weight: bold}

**Multiple Extends**

A single selector can extend more than one selector. This means that it inherits the styles of all the extended selectors. For example:

.error {

border: 1px #f00;

background-color: #fdd;

}

.attention {

font-size: 3em;

background-color: #ff0;

}

.seriousError {

@extend .error;

@extend .attention;

border-width: 3px;

}

is compiled to:

.error, .seriousError {

border: 1px #f00;

background-color: #fdd; }

.attention, .seriousError {

font-size: 3em;

background-color: #ff0; }

.seriousError {

border-width: 3px; }

In effect, everything with class .seriousError also has class .error *and* class .attention. Thus, the styles defined later in the document take precedence: .seriousError has background color #ff0 rather than #fdd, since .attention is defined later than .error.

**Chaining Extends**

It’s possible for one selector to extend another selector that in turn extends a third. For example:

.error {

border: 1px #f00;

background-color: #fdd;

}

.seriousError {

@extend .error;

border-width: 3px;

}

.criticalError {

@extend .seriousError;

position: fixed;

top: 10%;

bottom: 10%;

left: 10%;

right: 10%;

}

Now everything with class .seriousError also has class .error, and everything with class .criticalError has class .seriousError *and* class .error. It’s compiled to:

.error, .seriousError, .criticalError {

border: 1px #f00;

background-color: #fdd; }

.seriousError, .criticalError {

border-width: 3px; }

.criticalError {

position: fixed;

top: 10%;

bottom: 10%;

left: 10%;

right: 10%; }

**Selector Sequences**

Selector sequences, such as .foo .bar or .foo + .bar, currently can’t be extended. However, it is possible for nested selectors themselves to use @extend. For example:

#fake-links .link {@extend a}

a {

color: blue;

&:hover {text-decoration: underline}

}

is compiled to

a, #fake-links .link {

color: blue; }

a:hover, #fake-links .link:hover {

text-decoration: underline; }

**Merging Selector Sequences**

Sometimes a selector sequence extends another selector that appears in another sequence. In this case, the two sequences need to be merged. For example:

#admin .tabbar a {font-weight: bold}

#demo .overview .fakelink {@extend a}

While it would technically be possible to generate all selectors that could possibly match either sequence, this would make the stylesheet far too large. The simple example above, for instance, would require ten selectors. Instead, Sass generates only selectors that are likely to be useful.

When the two sequences being merged have no selectors in common, then two new selectors are generated: one with the first sequence before the second, and one with the second sequence before the first. For example:

#admin .tabbar a {font-weight: bold}

#demo .overview .fakelink {@extend a}

is compiled to:

#admin .tabbar a,

#admin .tabbar #demo .overview .fakelink,

#demo .overview #admin .tabbar .fakelink {

font-weight: bold; }

If the two sequences do share some selectors, then those selectors will be merged together and only the differences (if any still exist) will alternate. In this example, both sequences contain the id #admin, so the resulting selectors will merge those two ids:

#admin .tabbar a {font-weight: bold}

#admin .overview .fakelink {@extend a}

This is compiled to:

#admin .tabbar a,

#admin .tabbar .overview .fakelink,

#admin .overview .tabbar .fakelink {

font-weight: bold; }

**@debug**

The @debug directive prints the value of a SassScript expression to the standard error output stream. It’s useful for debugging Sass files that have complicated SassScript going on. For example:

@debug 10em + 12em;

outputs:

Line 1 DEBUG: 22em

**@warn**

The @warn directive prints the value of a SassScript expression to the standard error output stream. It’s useful for libraries that need to warn users of deprecations or recovering from minor mixin usage mistakes. There are two major distinctions between @warn and @debug:

1. You can turn warnings off with the --quiet command-line option or the :quiet Sass option.
2. A stylesheet trace will be printed out along with the message so that the user being warned can see where their styles caused the warning.

Usage Example:

@mixin adjust-location($x, $y) {

@if unitless($x) {

@warn "Assuming #{$x} to be in pixels";

$x: 1px \* $x;

}

@if unitless($y) {

@warn "Assuming #{$y} to be in pixels";

$y: 1px \* $y;

}

position: relative; left: $x; top: $y;

}

**Control Directives**

SassScript supports basic control directives for including styles only under some conditions or including the same style several times with variations.

**Note that control directives are an advanced feature, and are not recommended in the course of day-to-day styling**. They exist mainly for use in [mixins](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixins), particularly those that are part of libraries like [Compass](http://compass-style.org/), and so require substantial flexibility.

**@if**

The @if directive takes a SassScript expression and uses the styles nested beneath it if the expression returns anything other than false:

p {

@if 1 + 1 == 2 { border: 1px solid; }

@if 5 < 3 { border: 2px dotted; }

}

is compiled to:

p {

border: 1px solid; }

The @if statement can be followed by several @else if statements and one @else statement. If the @if statement fails, the @else if statements are tried in order until one succeeds or the @else is reached. For example:

$type: monster;

p {

@if $type == ocean {

color: blue;

} @else if $type == matador {

color: red;

} @else if $type == monster {

color: green;

} @else {

color: black;

}

}

is compiled to:

p {

color: green; }

**@for**

The @for directive has two forms: @for $var from <start> to <end> or @for $var from <start> through <end>. $var can be any variable name, like $i, and <start> and <end> are SassScript expressions that should return integers.

The @for statement sets $var to each number from <start> to <end>, including <end> if through is used. Then it outputs the nested styles using that value of $var. For example:

@for $i from 1 through 3 {

.item-#{$i} { width: 2em \* $i; }

}

is compiled to:

.item-1 {

width: 2em; }

.item-2 {

width: 4em; }

.item-3 {

width: 6em; }

**@while**

The @while directive takes a SassScript expression and repeatedly outputs the nested styles until the statement evaluates to false. This can be used to achieve more complex looping than the @for statement is capable of, although this is rarely necessary. For example:

$i: 6;

@while $i > 0 {

.item-#{$i} { width: 2em \* $i; }

$i: $i - 2;

}

is compiled to:

.item-6 {

width: 12em; }

.item-4 {

width: 8em; }

.item-2 {

width: 4em; }

**Mixin Directives**

Mixins allow you to define styles that can be re-used throughout the stylesheet without needing to resort to non-semantic classes like .float-left. Mixins can also contain full CSS rules, and anything else allowed elsewhere in a Sass document. They can even take [arguments](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixin-arguments) which allows you to produce a wide variety of styles with very few mixins.

**Defining a Mixin: @mixin**

Mixins are defined with the @mixin directive. It’s followed by the name of the mixin and optionally the [arguments](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixin-arguments), and a block containing the contents of the mixin. For example, the large-text mixin is defined as follows:

@mixin large-text {

font: {

family: Arial;

size: 20px;

weight: bold;

}

color: #ff0000;

}

Mixins may also contain selectors, possibly mixed with properties. The selectors can even contain [parent references](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#referencing_parent_selectors_). For example:

@mixin clearfix {

display: inline-block;

&:after {

content: ".";

display: block;

height: 0;

clear: both;

visibility: hidden;

}

\* html & { height: 1px }

}

**Including a Mixin: @include**

Mixins are included in the document with the @include directive. This takes the name of a mixin and optionally [arguments to pass to it](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#mixin-arguments), and includes the styles defined by that mixin into the current rule. For example:

.page-title {

@include large-text;

padding: 4px;

margin-top: 10px;

}

is compiled to:

.page-title {

font-family: Arial;

font-size: 20px;

font-weight: bold;

color: #ff0000;

padding: 4px;

margin-top: 10px; }

Mixins may also be included outside of any rule (that is, at the root of the document) as long as they don’t directly define any properties or use any parent references. For example:

@mixin silly-links {

a {

color: blue;

background-color: red;

}

}

@include silly-links;

is compiled to:

a {

color: blue;

background-color: red; }

Mixin definitions can also include other mixins. For example:

@mixin compound {

@include highlighted-background;

@include header-text;

}

@mixin highlighted-background { background-color: #fc0; }

@mixin header-text { font-size: 20px; }

Mixins that only define descendent selectors, can be safely mixed into the top most level of a document.

**Arguments**

Mixins can take arguments SassScript values as arguments, which are given when the mixin is included and made available within the mixin as variables.

When defining a mixin, the arguments are written as variable names separated by commas, all in parentheses after the name. Then when including the mixin, values can be passed in in the same manner. For example:

@mixin sexy-border($color, $width) {

border: {

color: $color;

width: $width;

style: dashed;

}

}

p { @include sexy-border(blue, 1in); }

is compiled to:

p {

border-color: blue;

border-width: 1in;

border-style: dashed; }

Mixins can also specify default values for their arguments using the normal variable-setting syntax. Then when the mixin is included, if it doesn’t pass in that argument, the default value will be used instead. For example:

@mixin sexy-border($color, $width: 1in) {

border: {

color: $color;

width: $width;

style: dashed;

}

}

p { @include sexy-border(blue); }

h1 { @include sexy-border(blue, 2in); }

is compiled to:

p {

border-color: blue;

border-width: 1in;

border-style: dashed; }

h1 {

border-color: blue;

border-width: 2in;

border-style: dashed; }

**Output Style**

Although the default CSS style that Sass outputs is very nice and reflects the structure of the document, tastes and needs vary and so Sass supports several other styles.

Sass allows you to choose between four different output styles by setting the [:style option](file:///C:\Users\igrove01\Desktop\file.SASS_REFERENCE.html#style-option) or using the --style command-line flag.

**:nested**

Nested style is the default Sass style, because it reflects the structure of the CSS styles and the HTML document they’re styling. Each property has its own line, but the indentation isn’t constant. Each rule is indented based on how deeply it’s nested. For example:

#main {

color: #fff;

background-color: #000; }

#main p {

width: 10em; }

.huge {

font-size: 10em;

font-weight: bold;

text-decoration: underline; }

Nested style is very useful when looking at large CSS files: it allows you to easily grasp the structure of the file without actually reading anything.

**:expanded**

Expanded is a more typical human-made CSS style, with each property and rule taking up one line. Properties are indented within the rules, but the rules aren’t indented in any special way. For example:

#main {

color: #fff;

background-color: #000;

}

#main p {

width: 10em;

}

.huge {

font-size: 10em;

font-weight: bold;

text-decoration: underline;

}

**:compact**

Compact style takes up less space than Nested or Expanded. It also draws the focus more to the selectors than to their properties. Each CSS rule takes up only one line, with every property defined on that line. Nested rules are placed next to each other with no newline, while separate groups of rules have newlines between them. For example:

#main { color: #fff; background-color: #000; }

#main p { width: 10em; }

.huge { font-size: 10em; font-weight: bold; text-decoration: underline; }

**:compressed**

Compressed style takes up the minimum amount of space possible, having no whitespace except that necessary to separate selectors and a newline at the end of the file. It also includes some other minor compressions, such as choosing the smallest representation for colors. It’s not meant to be human-readable. For example:

#main{color:#fff;background-color:#000}#main p{width:10em}.huge{font-size:10em;font-weight:bold;text-decoration:underline}