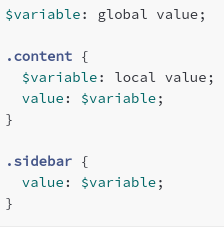
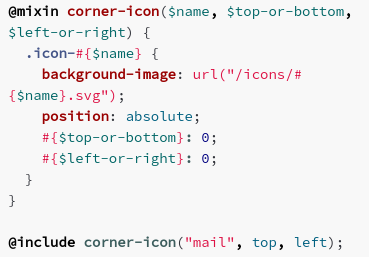
* [Online SCSS →CSS converter](https://codebeautify.org/scss-compiler) (may help with running experiments)

[Official SASS documentation](https://sass-lang.com/documentation/)

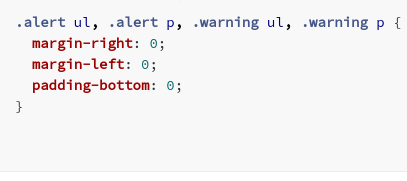
* [Special Functions (this is tough)](https://sass-lang.com/documentation/syntax/special-functions)
* [Defining variables (pretty basic)](https://sass-lang.com/documentation/variables)
  + Use the **$var-name: var-value;** syntax
    - SASS variables are compiled before conversion to CSS; CSS variables kept
  + Defining a variable later can override a previously-defined value. If you don’t want a variable to override previously-defined values, use the **!default** flag after the value. This allows variables to be configurable by variable overrides on the local scope. In the example below, the $time variable would be the first defined value, while $seconds will have the given value…
    - $time: 03312023;
    - $time: 04012023 !default;
    - $seconds: 222 !default;
  + You cannot override built-in variables (ex: the value of pi)
  + Put variables you want to be used throughout your program at the top to make them global. Variables defined in blocks/selectors will be local to that block. It is possible to override global variables within a block only by defining them locally: this is known as shadowing.



* + If you want a locally-defined variable to override a global variable, use the **!global** flag. For example, if we put the !global flag after the value of the $variable in .content, then this would become global and likely be applied to .sidebar
  + Built-in functions
    - **meta.variable-exists($var-name)**: Checks if a variable with the given name exists in the corresponding scope (local or global)
    - **meta.global-variable-exists($var-name)**: Checks if a global variable with the given name exists.
* [Interpolation (central concept!!!)](https://sass-lang.com/documentation/interpolation)
  + Very often used with @mixin. Mixins follow very similar purpose/formatting to functions in Python



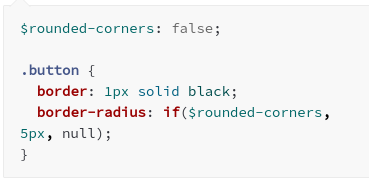
* + Interpolation: Wherever the parameter goes into the code, it is surrounded in **#{para1}**.
    - Note: Interpolation returns unquoted strings, so you can’t perform numerical operations. However, when defining a variable as a number, you can add units! There are some quirks there though.
    - Quotes around strings are removed when converting to CSS. However, using the built-in **string.unquote($string)** may be better for code readability purposes
    - On the other hand, if you wanted to preserve the quotes during interpolation, use the **#{meta.inspect($string)}** built-in function
* [Style notes (more thorough)](https://sass-lang.com/documentation/style-rules)
  + Nest a selector within a selector; the parent selector’s rules applied to the child selectors
  + Can list selectors



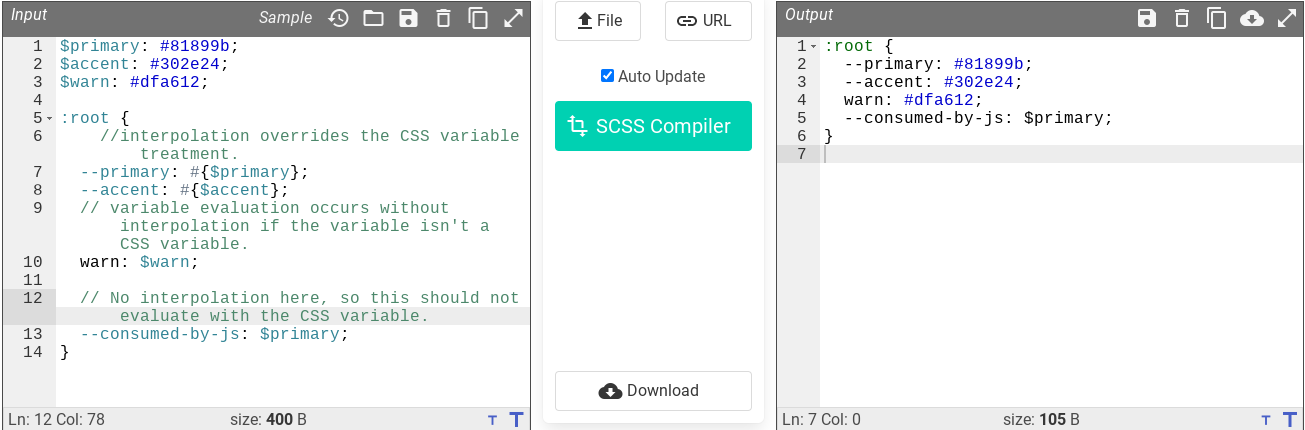
* + Can use combinators (I have no idea what that means)
  + Interpolation: Can make @mixin (form of procedural abstraction) and can allow user-inputted parameters to take effect
* [Property Declarations](https://sass-lang.com/documentation/style-rules/declarations)
  + Here is an example of a property being declared (.circle is property)



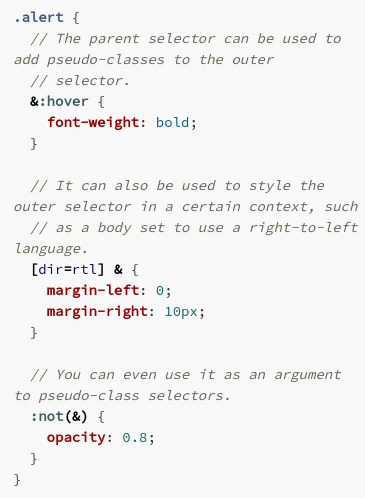
* + Can use [expressions](https://sass-lang.com/documentation/syntax/structure#expressions) for a declaration value, which can be evaluated using operators
  + Expressions include all the data types we’re familiar with (boolean, integers, etc.) The data structure known as dictionaries in Python is known as **maps** in SASS and use normal parenthesis instead of curly brackets.
  + Can manipulate using operators
  + Expressions include variables, calls to functions (including built-in [functions](https://sass-lang.com/documentation/syntax/special-functions))
  + Can use interpolation to define properties (reduces redundancy)
  + Can use nesting (covered above)
  + Hidden declarations: These are basically conditional statements. They’re called as such because the configuration won’t show up in CSS if the condition is not fulfilled. Look for the **if($boolean-condition, setting1, setting2)**



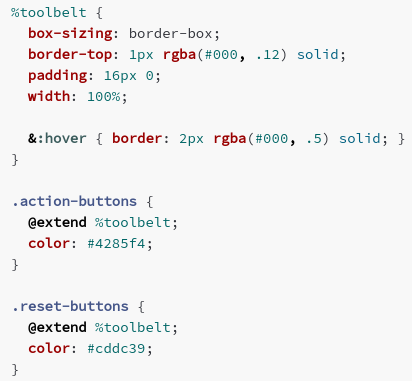
* + CSS custom properties: denoted by the double-dash before the variable name (ex:--var-name). If the value is set equal to a $string, the $string is not evaluated to a variable even if one is defined. Interpolation will override this. On the other hand, using one or no dashes for the variable name will result in evaluation of $string.
    - The documentation says this has implication for JavaScript compatibility



* [Parent Selectors](https://sass-lang.com/documentation/style-rules/parent-selector)
  + Whenever you use the **&** within a child class (in a nesting scheme), this is interpreted as the name of the parent class. Basic functions:
    - Adding suffixes to classes (ex: .parent\_\_child1 → &\_\_child1)
    - Adding a [pseudoclass](https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes), where the styles will change dynamically based on certain conditions



* + - There are some notes for advanced nesting I’m not ready for at the moment
* [Placeholder Selectors](https://sass-lang.com/documentation/style-rules/placeholder-selectors)
  + They are denoted using the **%** (ex: **%button-settings**). They are not translated directly into CSS. Instead, you can use **@extend** to import and combine all the styles in placeholder selectors into selectors of your choice.



[List of at rules (this will be somewhat comprehensive)](https://sass-lang.com/documentation/at-rules)

* [@use](https://sass-lang.com/documentation/at-rules/use) - essentially a replacement to @import
  + Loads all mixins, variables, functions (referred to from here on out as **members**) from the files. Any CSS output that would come from running those files normally could be outputted here. The file will only be loaded once regardless of how many times @use is called
    - Variables/mixins from files usually have namespaces to prevent conflicts (**namespace.var-name**). By default, the name space is the file name, though this can be customized. Using \* will result in no namespace. @import does not have this option!
    - Great if you got a scheme with multiple .scss files. Each .scss file loaded is now known as a **module**. Use @use before any other at-rule in the file.
      * If you don’t want the SASS of a loaded file to be compiled/interpreted individually, start the file name with a **\_** (the file is now known as a **partial**)
    - Can use variables with **!default** flags in the original file to allow for greater flexibility
  + SASS loads the URL (not the file path). This is for cross-compatibility between Windows (with backslashes) and Linux (with slashes)
    - Need to do **@use ‘url/file’;**
    - Putting the file extension is optional. Can load .scss, .sass, and .css files. .css files can’t use .sass features, but .css can be extended



* + Need to look into integration with @mixin
  + It is possible to re-assign variables after the @use is called (but not built-in variables)



* + If there is an \_index.scss, then this is loaded when @use is called on the URL of the folder.
  + Deviations from @import (the predecessor)
    - All the variables, functions, mixins only to the scope of the file (not at global scope)
    - Each file loaded once (reduces redundancy)
    - Must be at the top of file, with one URL with quotes around it.
  + May exclude variables, mixins, functions by starting the name with an underscore/hyphen (ex: **$-radius**). When @use is called on a file, those members are not loaded and are known as **private members**. This could help with controling what parts of the file are loaded into another file (ex: sensitive info)
* [@forward](https://sass-lang.com/documentation/at-rules/forward)
  + Very similar to @use in that it loads the members, making them available as if they were in the module (they aren’t though; that’s what @use is for). With regards to CSS, it functions the same way as @use.
  + If you want to combine with @use, do @forward first.
    - Useful if you want to load all the variables from one file
  + Can add variable prefixes to avoid conflicts using **@forward “url” as prefixname-\***

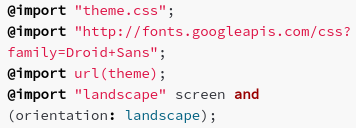
****

* + Can do **@forward “url” hide <members>** to exclude specified members or **@forward “url” show <members>** to include only specified members
  + Can combine with variables of the **!default** flag



* [@import](https://sass-lang.com/documentation/at-rules/import)
  + **This will eventually be deprecated**
    - All members defined globally → where did they come from?
    - Naming collisions for members is a concern since namespaces aren’t added by default
    - Inefficient output (stylesheet converted to CSS upon every import instance, as opposed to once when calling @use. Could result in redundant code)
    - No compatibility for private members
  + Functionality: All the file’s members & CSS results replace @import
    - Similar to @use, do the URL first. If you want to load based on paths, you can load relative to current file (in same directory as file directory or given load path). Can use commas in the @import call (see below).
    - Has all the same functionality as @use with regards to which files are compatible, partials, index files, loading CSS files
    - Possible to name your file **filename.import.scss** to make sure it’s only @import compatbile (not @use compatible). If using a file meant for @use, the @import has access to private members but not other loaded modules unless @forward is used. Modules can only be loaded once, meaning the members can’t be changed.
    - Since @import was meant to be an improvement on @import from CSS, meaning that @import in CSS had to be supported. Imports with a .css extension, a URL with the http:// or https://, url(url.css), and media queries (see below). Also, CSS @import can include interpolation (ex: based on parameters of mixins; SASS @import can’t have interpolation)





At this point I realized that the concision of my notes was not optimal. I’d probably be returning to the documentation anyways, so I’ll stop providing example code snips here. My notes from here on out will take a more conceptual or feature-focused approach, so if you’re interested in including the features in the lesson, take a gander at the docs (it’ll help you get used to docs, which you’ll be consulting a lot)

* [@mixin, @include](https://sass-lang.com/documentation/at-rules/mixin)
  + Allows for definition of re-usable styles based on parameters. Define using **@mixin mixin-name** at the top of the file.
  + Use **@include mixin-name (arguments);** to use the mixin. It is possible to make arguments optionalto input by defining a default value in the mixin using **$var-name: default-value**. All arguments must be utilized in the code, however.
  + It is possible to pass arguments during a mixin call using keywords using similar syntax to optional arguments in the call. This can be useful if the user wants to be sure of how variables are being configured. If you rename an argument, you should tell the users and keep the old name for some time to give them time to adjust.
  + You can take the last arguments as a list by doing: **$arg…** All the extra arguments then become elements of a list, and you can do iteration operations accordingly. This also allows you to pass a list variable as the last argument. These arguments are known as **arbitrary arguments**.
    - If users like keyword arguments in their calls, you need to **@use “sass:meta”;** at the top and call **@debug meta.keywords($arguments);**. This converts the user input to a map which allows the code to work with the input.
  + There is an @content rule that is nowhere else in the docs. If you include it in a mix-in, it becomes a **content block.** Content blocks have lexical scope, meaning it will only see its own local variables (and not any of the ones locally defined in the mixin where it is called). For mixins with content blocks, you **must** declare the acceptance of the arguments (**@include mixin(para1, para2) using ($argument) {** )
* [@function](https://sass-lang.com/documentation/at-rules/function) - **@function function\_name (arguments) {code}**
  + Very similar in function to mixins in that this serves for procedural abstraction. The docs encourages use of functions for computing values and mixins for setting variables dynamically
  + Can only have [universal statements](https://sass-lang.com/documentation/syntax/structure#universal-statements) in the code (declaring variables, @error, @warn, @debug, at rules for flow control)
  + Identical functionalities to mixins with regards to default values (optional arguments), final arguments as a list (arbitrary arguments), passing arguments using keywords
  + Different in that functions are called using their names and that there must be an **@return** to exit the function with the indicated value.
  + There are built-in SASS functions (more on those later). Also, you can call CSS functions, which are not user-defined or built into SASS.
* [@extend](https://sass-lang.com/documentation/at-rules/extend)
  + Call **@extend .selector-name** to make the given selector inherit the style/element rules as the extender referenced. Best to use for semantic (related) selectors/classes
  + @extend updates/overrides any styles as appropriate and reduces redundancy through combination of selectors with the same attributes.
  + This will also function for any pseudoclass where the given selector exists.
  + Since these are compiled towards the end, parent selectors are blind to @extend effects
  + Can put placeholders (selectors purely developed for extension) using the %, could be private by starting the name with - or \_ (can only be extended within its stylesheet)
  + @extend usually has limited scope (only affecting its stylesheet and ones that are loaded with @use or @forward). The exception is @import, where @extend is imported on a global scale.
  + @extend requires that the selector exists unless the **!optional** flag is added. In that case, @extend does nothing if the selector doesn’t exist
  + You can’t extend complex selectors like **.complex.selector** the same way as usual. You must extend using **@extend .complex, .selector**. If complex selectors are interleaved, @extend uses a heuristic to avoid using combinations that will not exist in HTML.
  + If you want to use an **@media** (CSS at rule), can’t extend selectors outside of @media
* [@error](https://sass-lang.com/documentation/at-rules/error) - **@error <expression>**
  + Yeah, this is in case your user gives an unexpected argument or if the argument passed is otherwise unexpected. This could be the case with typos or with deprecated arguments. You give an error message and the stylesheet quits compiling. Probably good for ensuring the integrity of your data structures.
* [@warn](https://sass-lang.com/documentation/at-rules/warn) - **@warn <expression>**
  + This discourages users from entering unusual arguments. It has a similar printing functionality to @error, but it doesn’t quit the compilation of SASS.
* [@debug](https://sass-lang.com/documentation/at-rules/debug) - **@debug <expression>**
  + This is how you get to see the values of your variables (identical functionality to **print()** in Python). As the name implies, useful for finding out what your program is actually doing.
* [@at-root](https://sass-lang.com/documentation/at-rules/at-root)
  + Everything with regard to the selector referred to in the at-root is compiled at the bottom of CSS rather than following usual nesting rules. This may be required for certain nesting schemes.
    - Usually gets rid of style rules only (not CSS rules). Can exclude certain rules with **@at-root (without: rulename)** or include only certain rules using **@at-root (with: rulename**)
    - **Rule** is style rules but not at rules.
    - **All** is all style rules and all at rules.
* Flow control rules: These are essentially selection and iteration. The two ways for loops index in Python are separated in SASS.
* [@if and @else](https://sass-lang.com/documentation/at-rules/control/if) - **@if <expression> { code }**
  + You guessed it, this is selection. The code associated with @if is evaluated when an associated expression is true, while that associated with @else is evaluated if the expression is false. If you want a chain of conditionals, use **@else if**
  + Truthy: Any expression that is not **false** or **null** is true. This includes empty strings/lists and **0**.
* [@each](https://sass-lang.com/documentation/at-rules/control/each) - **@each <var> in <expression> { code }**
  + Use this when you want to execute code for each element in a list or pairs in maps. For lists, you can assign one variable name for the elements, but in maps, you must have two variable names (one for the keys, and one for the corresponding values)
  + Lists of lists are a thing, and if you happen to be working with one you have to assign one variable for each element index (ex: if each list has 4 elements, you must define 4 variables in the @each call)
* [@for](https://sass-lang.com/documentation/at-rules/control/for) - **@for <variable> from <expression> to <expression> {code}**
  + This is for executing a block of code a certain number of times. The expressions are essentially indexes, and SASS counts from the first expression to the second expression. Using **to** makes the final index excluded; use **through** to include this.
* [@while](https://sass-lang.com/documentation/at-rules/control/while) - **@while <expression> {code}**
  + Will evaluate the code block repeatedly until the expression evaluates to false. Remember the truthy and falsy rules in SASS!
    - The documentation recommends using @each and @for over @while where possible
* [CSS rules](https://sass-lang.com/documentation/at-rules/css) - support for all rules
  + If CSS nested inside style, the CSS at rule is outputted on top for ease of use
  + Can use @media for queries. According to [Mozilla](https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries#targeting_media_features), this allows for styles to vary based on conditions like screen size. Can use interpolation and use SASS expressions. Nested queries will be merged since some browsers don’t support nested queries.
  + [@supports](https://developer.mozilla.org/en-US/docs/Web/CSS/@supports) is used to apply styles if the machine supports such styles.
  + [@keyframes](https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes) is used to apply styles to intermediary steps in animated portions of a website.
* [Values](https://sass-lang.com/documentation/values): These are basically all the data types supported in SASS.
* [Numbers](https://sass-lang.com/documentation/values/numbers)
  + Can have units and scientific notation (ex: 1.2e4 = 12000). SASS fully expands the numbers since browsers do not consistently support scientific notation.
  + SASS can do unit manipulations, though if you have complex units (ex: px2), using those as property values will produce errors. This feature is intended because usually complex units indicate calculation errors.
  + SASS can convert units, but if you do operations with incompatible units there will be errors. Use **math.comparable($number1, $number2)** to see if the units are compatible.
  + Do not use interpolation if you want a number, because interpolation gives a string without quotes.
  + Decimals and percentages are not interchangeable unless using **math.div($percentage, 100%)** or **math.percentage()**
  + Has 10-decimal precision. Numbers with negligible differences reaching more than 10 decimals will be considered equivalent, and there are certain rounding conventions (rounding tends to gravitate towards integers where applicable)
* [Strings](https://sass-lang.com/documentation/values/strings)
  + Can be quoted or unquoted. Can convert between these forms using **string.unquote()** and **string.quote()**
  + Can use backslashes to make characters other than the hex characters part of a character. Could alternatively enter the unicode for any character, and the unicode is replaced by the corresponding character.
  + Quoted strings can have single quotes or double quotes. Can have interpolation and contain most characters (\, ‘, “, and newlines must be escaped with unicode)
    - Guaranteed compilation. At this point we see why interpolation removes quotes (so that quoted strings inserted within another quoted string don’t cause issues)
  + Unquoted strings follow a syntax pattern (on-site). It may be a good idea to include dashes or underscores. Some identifiers not parsed as strings are:
    - Color names → colors
    - null → the value null
    - true/false → the boolean values
    - not, and, or → boolean operators
  + Escapes is weird for unquoted strings. Unicode is converted where appropriate, and characters other than newlines and tabs can be replaced (including the backslash). Sometimes, there is no compilation (especially if non hex characters are involved in the backslash). Honestly I think it’s best to use quoted strings for simplicity
  + Each character in a string is indexed. A few quirks:
    - The first character has index 1 (rather than 0). Yep, just like pseudocode.
    - The last character can have index -1, the second to last having -2, etc.
* [Colors](https://sass-lang.com/documentation/values/colors)
  + Supports: sRGB, HSL, colors as hex codes, color names
    - Any color can be used with both RGB and HSL)
  + There are built-in functions to modify colors. More on that later
* [Lists](https://sass-lang.com/documentation/values/lists)
  + Elements can be separated with slashes, commas, or spaces (if consisted)
    - Forward slashes are also used for division, but this will be deprecated in favor of **math.div()** in the near future (outside of the **calc()** function). This will make forward slashes only function as list separators.
  + Brackets are optional. Use square brackets or parenthesis
    - Can have one-element lists **(expression,)** or **[expression]** or zero-element lists **()** or **[]**
  + Remember that we are using the 1-index (not 0-index)
  + Get an element: **@debug list.nth($list, $index)**
  + Use @each to do something for each element
  + To append: **@debug list.append($list, $appended-value)**
  + To check if a value exists: **@debug list.index($list, $value)** (returns the index of the value or null if none)
  + Lists are immutable (can’t be modified); all the list functions make new lists. It is possible to assign the new list to the same variable and override, however.
  + If you make the last parameter of a mixin with …, you can take the last argument as a list.
* [Maps](https://sass-lang.com/documentation/values/maps)
  + **(<key>: <value>, <key>: <value>)**
    - Keys must be unique. Duplicate values are OK. Put keys in quotes to make sure you’re actually making them string values (and not some other value, like a color value).
    - Can use == to determine if keys have the same value
  + Lookup a value: **@debug map.get($map, $key)**
  + Use **@each** to execute code for each key and value pair
  + Add/update key/value pair: **map.set($map, $key, $value)**
  + Merge maps: **map.merge($map1, $map2)**
    - Duplicate keys: Second map’s values override first map’s values.
  + Maps are immutable because they are lists.
* [True/False](https://sass-lang.com/documentation/values/booleans)
  + Are booleans. Can be returned by operators or built-in functions.
  + Can be affected by boolean operators (and, or, not, etc.)
  + Can use booleans in conditionals
  + Remember: only **false** and **null** are false.
* [Null](https://sass-lang.com/documentation/values/null)
  + Returned by functions when there is no result.
  + Properties and list elements with **null** are omitted from CSS during compilation.
* [Calculations](https://sass-lang.com/documentation/values/calculations)
  + Use built-in functions (**calc(), min(), max()**) to do calculations. Using / for division will work here.
    - Functionally equivalent to CSS **calc()**, but SASS variables and SASS functions can be used.
  + Calculation functions can also simplify. Occasionally, simplification is prioritized over calculation
  + Calculations can cause errors since some operations can only be used on numbers without units
  + Constants can be used. Pi, e, infinity, -infinity, and NaN (not a number) are resolved/parsed automatically.
* [W3Schools](https://www.w3schools.com/sass/sass_intro.php)
  + One of the biggest selling points on the front page of W3Schools SASS is assigning colors (ex: #ffffff) to variables
  + Fonts, text size, and image size can also be assigned to [variables](https://www.w3schools.com/sass/sass_variables.php), as can lists/booleans (like in most programming languages)
  + Can [nest](https://www.w3schools.com/sass/sass_nesting.php) variables in SASS (rather than spelling them out one-by-one in CSS)

What is SASS:

* SASS code needs to later be converted into CSS code as browsers can’t read SASS code
  + A compiler can be used for this conversion
* SASS is a lot more concise than CSS and therefore more scalable.
  + [Alert](https://www.w3schools.com/sass/sass_mixin_include.php): Hyphens and underscores are considered the same character.

Features available with SASS:

* **Variables**: Variables can be created using a $ sign to store things like different colors that can be referenced in different parts of the SASS code
  + If the thing stored in the variable needs to be changed, it only needs to be changed in the variable rather than every place the thing is used (without SASS) which would be a lot of time consuming
* **Nesting**: Nesting can be used to avoid different code segments being repeated various times. Also increases ease of variable definition (ex: font-related variables)
  + [Selector Lists](https://sass-lang.com/documentation/style-rules#selector-lists): Selectors that are between commas are nested separately and combined in CSS. Can be combined with [combinators](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors#combinators). May have quadratic complexity (watch it!)
* **SASS Mixins**: If you find yourself wanting to use the same group of styles for many different classes, the mixins feature can be used to apply a style set anywhere you want
* **Conditional Logic**: If and else statements can be used with SASS.
* **Lists** and **Arrays** can be created with SASS
* **Loops**: For, while, and each loops can be implemented with SASS
* **Functions**: Reusable functions can be created and the last 3 features mentioned before can be used in them.
* [Importations](https://www.w3schools.com/sass/sass_import.php): Use **@import filename.scss** to import a file and all the variables defined within that file. This allows creating a global cache of variables for the project. This prevents redefinition of variables for every file (that’s redundant!)
* [Extensions](https://www.w3schools.com/sass/sass_extend.php): Another way to increase concision in SASS. You can define a class with some attributes (text size, text alignment, etc.) as a generic class (ex: buttons on your website will be visually similar). In another class, call the generic class using **@extend .class-name;** and all the attributes of that class will be imported into your defined class.

Syntax:

* SCSS:
  + File extension is .scss
  + Use this because it is more popular
  + All CSS works in SCSS too
  + Uses curly braces and semicolons (whitespace does not matter)
* Indented syntax:
  + File extension is .sass
  + Same features as SCSS except indentation is used (whitespace does matter)

Example Code:

* [FastPages (Leonard’s posted here)](https://github.com/Leonard514/FastPage/tree/master/_sass/minima)
  + dark-mode.scss and fastpages-dracula-highlight.scss has good stuff
* Check Lily’s fastpages for SASS examples (will post later)

Possible Quiz Questions:

* What does SASS stand for?
* Give three reasons why SASS is more scalable than CSS.