%%
Contents 7ii Produce Maintainable CSS With Sass **Panteli**
%%

%%
7ii Part #2 - Create efficient, maintainable code with intermediate Sass techniques
%%

1. Improve code maintainability with Sass variables

- **video notes**
  - -> Sass is to make the code more convenient
  - -> Saas variables
    - if the same parameters are used again and again -> then assign their values to a variable name and reuse the name of the variable throughout the course
- **notes on the text below the video**
  - Leverage
    - -> you can't just work hard, you have to work hard at the right approach
    - -> leverage <- use the right approach and for the same amount of effourt - you can get the same results
    - -> this is the idea of variables in Sass
  - Set it and forget it
    - -> the site uses the same colours again and again <- repeating the same hex codes

        - setting variables equal to the colour hex codes, then reusing them
            - -> if the client requests a colour change
            - -> changing the same colour manually on repeat creates the possibility of errors
    - -> naming variables according to their function
        - declaring a Sass variable is setting its value -> $variable-name
            - -> e.g -> $mint: #15DEA5;
                - this is how you define the variable name in Sass
                - -> then to use it -> for example you would replace #15DEA5 in the code with $mint (the variable name which stores the colour)
            - -> there is no css equivalent of this, if you try and compile the Sass into css
                - then when you use the variable name in the Sass -> and compile it into css, then the value which the variable stores will be shown in the css (rather than the variable name)

```sass
$mint: #15DEA5;

.form {
  width: 100%;
  padding-bottom: 1.5rem;
}
.form__heading {
    width: 100%;
    color: #fff;
    text-shadow: 0.55rem 0.55rem #11af82;
    background: $mint;
    line-height: 5rem;
    padding: 1.5rem;
}
.form__field label {
    color: #D6FFF5;
    display: block;
    font-size: 2rem;
    line-height: 2rem;
    padding-top: 1.5rem;
}
.form__field input {
    width: 100%;
    background: #001534;
    border: 0.1rem solid $mint;
    padding: 1.5rem;
    color: #D6FFF5;
```

At the top -> setting the name of the variable, then using this name in the Sass, rather than the hex codes which it stores

    - -> use variables in Sass but not in css
        - -> css has a feature called custom properties which works like the variables in Sass
        - -> but this is messier syntax, so it's cleaner to use the regular css syntax and then the variables for Sass
    - -> making changes with variables
        - -> you can change the value which the variable stores
        - -> but then if its a colour -> e.g mint-green and it changes to pink, then you want to change the variable name along with it
        - -> name the variable according to its role / purpose, because the appearance of it can change -> then you have to go back and change everything
            - -> or e.g $colour-primary
            - -> when you are naming the element -> naming it with its appearance changing in future in mind
        - -> vermillion is red
    - -> task
        - -> replacing Sass hexcodes with variable names
        - -> then making sure that nothing in the appearance of the webpage had changed
        - -> again the exercise is in a codepen
            - -> the frequent use of codepen to trial new html / css / Sass (indented css)
    - -> eight data types in Sass <- colours, strings, numbers, lists, maps, other (booleans / nulls / function references)
        - -> this course only deals with the first five
        - -> there is Sass documentation
        - -> next is mixins <- variables for storing multiple css styles
- Let's recap!
    - -> Sass variable syntax is $variable-name
    - -> these are used to store the value of different css / Sass styling

2. Use SASS Mixins with arguments

- **video notes**

- -> repeating colours / properties
- -> e.g borders, text shadows
- -> **variables store one line of css, mixins are the variable equivalent of multiple lines of css**

- **notes on the text below the video**
  - Meet the variable's big brother
    - -> everything he's mentioned here is the same as in the video for the section
    - -> @ <- the syntax for mixins
  - Mixin it up
    - -> **mixin syntax**
      - **@mixin mixin-name {**
      - **   css-property: value;**
      - **}**
      - -> and under the css-property: value; <- you can store multiple different lines of css
      - -> you name according to the purpose not the appearane, because the appearance can change
    - -> he's using this in Sass <- so the css is indented, with the BEM naming convention -> so e.g form__heading (__ is a dunder)
    - -> when you use variables, you are defining them at the top of the css and then using them throughout it
      - -> this is defining multiple lines of css in a variable, and then using that throughout the rest of the Sass, as you would a normal variable
      - -> to call a mixin, it's @include name-of-mixin; in a block of Sass (indented css in a .scss file), and then to define it it's the css above
      - -> the equivalent css is to indent the mixin (to go from css to Sass, this is to compile the Saas in css -> when you do this for mixins in Saas, then the equivalent css is the styles which the mixing represented - replaced wherever the name of the mixin was used in the Sass, like it is with a variables, except multiple lines of css styling had been used there)
    - -> exercise
      - -> using borders on the site
      - -> mixins to avoid reusing the code for the borders
      - -> @mixins
      - -> when you add a mixin in, you need to make sure that the appearance of the webpage remains the same
      - -> a webpage is rendered html <- the rendering is done by a browser
  - Getting argumentative
    - **-> mixins can have arguments**
      - **-> so Sass is like indented css, in a .scss file**
      - **-> Sass can e.g store the names of colour hex codes in variables, which are defined once and then whose names are used throughout the Saas**
      - **-> if you have multiple lines being stored in one of those variables, then we're calling it a mixin**
      - **-> if you put the mixin into code which is e.g the header, and then put in into a paragraph in another case, you might want it to behave differently**
      - **-> that is where you give the mixin arguments**
      - **-> so we have multiple lines of css, and depending on which element is being targeted, then we have different lines targeting that element -> the element is the argument**
      - **-> arguments are done for minix using 'inputs' -> they are arguments**
    - -> syntax for mixin arguments

- in the definition of the mixin argument (example)
    - -> @mixin heading-shadow($colour){
    - text-shadow: .55rem .55rem $colour;
    - }
        - -> in this example, $colour has the syntax of a variable ($ refers to the value of this variable)
        - -> it's like an argument to a function - and in the declaration of the mixin, that's when the arguments are declared
        - -> they are then reused throughout the code which defines it
- in the use of the mixin argument
    - -> .heading{
    - &__header {
    - @include heading-shadow(#fff);
    - }
    - }
        - -> in this example: we have @include <- this call the mixin
        - -> then the name of it
        - -> then the argument
        - -> in the definition of the mixin, this argument was the colour
        - -> so when it's used, the argument its taking in this case is the hexcode of the shadow which it wants
        - -> and again, it's in scss (Saas, indented css)

- By default
    - -> setting default values for mixin arguments
    - -> so we have arguments (you can set specific values for something)
    - -> this is how you can set default values for those arguments
    - -> a mixin is like a function in Sass (indented css) with multiple lines of css, which styles then
        - -> and the function takes arguments (e.g the variable names for the different styles which is targets and which you can change)
        - -> then this section of the chapter is how you can give the arguments of those functions (mixins) default values
            - thinking about the Python analogy
    - the syntax
        - @mixin heading-shadow**($colour: $colour-primary**){
        - text-shadow: .55rem .55rem $colour;
        - }
        - -> in other words, the default value of the colour variable is $colour-primary
        - -> having a default value stops you from having to declare one each time, unless something else is used for that property
        - -> when you don't give it an argument when calling the mixin, then it will use this default value
    - -> exercise
        - -> when the colour of an element changes -> this is when he's added in an argument into the mixin to adapt to that change <- the idea of arguments with mixins being seen as a way of adapting the styling
        - -> we're adding an argument to the mixin <- the argument takes a colour
            - because in this example we're trying to adapt it to a situation with two different colour types
            - -> one of the colours is a default -> one of the two which is most commonly used
        - -> the thought process for adding an argument into an existing mixin is

        - -> add in $argument-name into the line which declares it
            - and a default for it
        - -> inside the definition, replace the hex codes with that $argument-name
            - in this case it's the hexcodes
        - -> then change the scss (Saas, indented css) -> change the times the mixin, aka function for styling has been used, to code which passes those arguments when its used
    - -> commonly - check to see that the appearance of the webpage hasn't changed
        - -> there can be several webpages which look the same but have different css defining them (which is structured differently)
        - -> many different solutions to the same thing
        - -> another common thing he does check it in codepen
- Circling the wagons
    - -> arguments in mixins
        - -> these can be used to change the colours
        - -> or to e.g have default colours
    - -> what you don't want the mixins to contain
        - -> e.g for the size of shadow offsets
        - -> things which are too specific to certain elements -
        - -> the entire idea of -> breaking down mixins (variables with multiple lines of css, into individual variables)
    - -> what he's doing is separating a variable out from a mixin to make the code less specific and more reusable
        - -> you can also make mixins more generic by defining more arguments in the mixin
        - -> **what he's done is set the default value of an argument in a mixin equal to the value of a variable which he's declared higher up on the page (sccs aka Saas aka css with indentations)**
        - -> below - this is it
            - $heading-shadow-size: 0.55rem; <- setting the value of the variable
            - @mixin heading-shadow($colour: $colour-primary, $size: $heading-shadow-size;){ <- including that variable the default value of a mixin argument
            - text-shadow: $size $size $colour;
            - }
            - -> that way the outcome is flexible and it's easier to maintain, because the default value is equal to another variable - whose value is declared at the top of the page / above it
    - -> best practices
        - -> colour and size values should be contained in variables
        - -> and then those variables be used as the arguments to mixins
        - -> the best practice is to store all of the values as variables, and then set the names of those variables equal to the default values for mixin arguments
        - -> and then in the definition of the mixins, those arguments are used in the form of $'s (values)
    - -> the next chapter is Sass extensions <- for duplicating modules of code
- Let's recap!
    - -> mixins are like variables which store multiple lines of css
        - literally they're like functions (they can have arguments, with default values)
            - scss, Saas = indented css
            - -> and that can include functions, aka mixins with blocks of css which are reused
            - -> they can have arguments to change the values of certain parameters
            - -> and then those get reused in the code and can take default values

                - those default values are set equal to the name of some variable which is declared at the top of the scss / Sass document
            - -> the arguments chang eht output of the compiled code -> the outputs
        - -> they are declared using @mixin and called using @include
        - -> for Sass which is the

3. Write cleaner code with Sass extensions

- **video notes**
    - -> mixins are good from having to stop repeating code
    - -> Sass can use extensions
    - -> these are similar to mixins in the outcome, but get the results in a very different way
- **notes on the text below the video**
    - sass extensions
        - -> why use extensions and not mixins
            - each time a mixin is used a line of css is repeated
            - -> e.g @include mixin-name; <- this one line is repeated again and again in the code
            - -> you have to define mixins in the scss file (Saas / css with indentations file)
            - -> every time you use the mixin, in the compiled css the code is repeated
                - -> the lines of css which define the mixin
                - -> this is what is repeated in the compiled css every time you use the mixin
            - -> using mixins messes with BEM naming conventions
                - i.e defining the value of a variable
                    - and then using it in the name of the mixin
                    - and reusing that line of code again and again
            - -> Sass extensions
                - these are blocks of code which can be reused
                - -> using Saas to reuse mixins <- compared to Saas extensions, mixins don't need to be declared with a special identifier
                - -> when extensions are defined -> they are defined as literal @extend's <- in the indented css, aka sass
                - -> you can write @extend inside a Sass class, you don't have two write an entirely new section for it like you would with a mixin
                - example
                    - .typography {
                    - color: $colour-primary; **<- the extension is defined in the sass (aka css with indentations) document at the top like a mixin would be, and then is called within css (Sass) classes later in it**
                    - font-size: 2rem;
                    - font-weight: 100;
                    - line-height: 1.7;
                    - }
                    -
                    - h1 {
                    - @extend typography; **<- to use the extension, it's referred to as @extent name-of-extension in a css class**
                    - }
                - -> exercise
                    - why use extensions and not mixins in Sass
                        - -> when you move from Sass to css, Sass is css with indentations in a .scss document and css is a styles.css document
                        - -> when you compile the Sass to css - you want to reduce the amount of repeated code

- -> if you define extensions in Sass, every time they are used, then the code which defines their styles isn't repeated in the compiled css
- -> but this isn't the case if you use mixins - every time the mixin is called in Sass -> this repeats the code for the styles which define it in the compiled css
- -> this is the advantage which Sass extensions have over mixins

- in this example
  - -> there are multiple block types on a webpage, and those elements have things in common
  - -> using extensions to link them
  - -> rather than mixins
  - -> and then you would separate out the parameters which they share into variables
    - -> the reason we use Sass extensions and not mixins is because when the mixins compile into css, each time you use them then the styles which define them get repeated in the compiled code
  - -> nesting and the & to create heading elements for the blocks <- so that the specificity doesn't increase more than it has to
  - -> @extend to extend new form elements to the other Sass classes
- -> at the end of the exercises, the rendered HTML is always reviewed in the browser to make sure nothing major has changed

- Using placeholders
  - -> what 'extensions' are doing in Sass, for the compiles css is -> it's like this
    - class1, class2 {
    - }
    - -> it's taking one class and 'extending it to' another
    - -> literally listing the styles for them, 'extending' its styles to it
      - rather than entirely duplicating the css for it which is what a mixin does
        - -> mixins are copying and pasting the entire css code
        - -> extensions are copying and pasting the name of one of the selectors

scss

```scss
.typography {
    color: $colour-primary;
    font-size: 2rem;
    font-weight: 100;
    line-height: 1.7;
}

h1 {
  @extend .typography;
}
textarea {
    @extend .typography;
}
button {
    @extend .typography;
}
input {
    @extend .typography;
}
```

css

```css
.typography, h1, textarea, button, input {
  color: #15dea5;
  font-size: 2rem;
  font-weight: 100;
  line-height: 1.7;
}
```

- -> (above) this is how Sass extensions compile into css -> they literally start listing all of the classes by that css styling
  - rather than repeating the styling itself

- **-> css placeholders -> instead of a .class {**
- **}**
    - **-> that he's done is %class {**
    - **}**
    - **-> in case that class isn't used**
    - -> a class which has been defined but might not be used
    - -> this % is called a placeholder
        - silent classes
        - placeholder classes
        - -> **think of the % like - if this thing isn't being used, then it takes up 0% of the css document**
    - -> you can use extensions on these classes
    - -> they are mostly just like normal classes -> you can @extend %.... from another class
    - example
        - %typography { <- the .class -> instead of this, in Sass for this example it's %
        - color: $colour-primary;
        - font-size: 2rem;
        - font-weight: 100;
        - line-height: 1.7;
        - }
        -
        - h1 {
        - @extend %typography; <- then to use the placeholder class -> it can be extended from in other classes like this
            - -> the name of it is refereed to as %name-of-class, not name-of-class
        - }
- exercise
    - -> in this example, there is a placeholder class
        - -> **think of a search bar <- before you start typing and click on the search bar, there is text there**
        - **-> the text - before it goes away when you click on the search bar <- this text is called placeholder text**
        - **-> so if we have placeholder classes**
            - **-> then these are the classes which 'go away' when they aren't being used, even if they have been defined**
    - -> there is a - in this example - placeholder class which isn't being / hasn't been used
        - -> but it is extending to other classes which are being used
        - -> the entire purpose of it is for it to be extended to other elements
        - -> rather than for it to be directly used
        - -> the context under which he has changed a class from a class .class, to a placeholder class %class, is the case where the class in Sass (indented css) isn't directly being used, but there are other elements which are @extending off of it
            - -> these elements all share parts of that Sass in common
    - -> again, the last stage is always to check that the webpage looks the same after than css has been changed to make it more efficient
    - -> using codepen again to ensure this is the case
    - -> the next chapter is mixins vs extensions
- Let's recap!

        - -> Sass extensions are similar to mixins
        - -> both are there to prevent code from being duplicated
        - -> **when mixins are defined and used in Sass, then every time they are called, in the compiled css this creates a duplicate of the rules which defined them**
        - **-> this is the opposite for selectors, where the selectors themselves (rather than the rules which define them) are duplicated**
        - -> to make sure you don't unused selectors (Sass classes = indented css classes)-> you can use placeholder rulesets %'s
            - %extend-placeholder
            - -> if you change a .class to a %class, all of the elements which are extending to it have to have that % put in, whenever the (now placeholder) class is being extended to

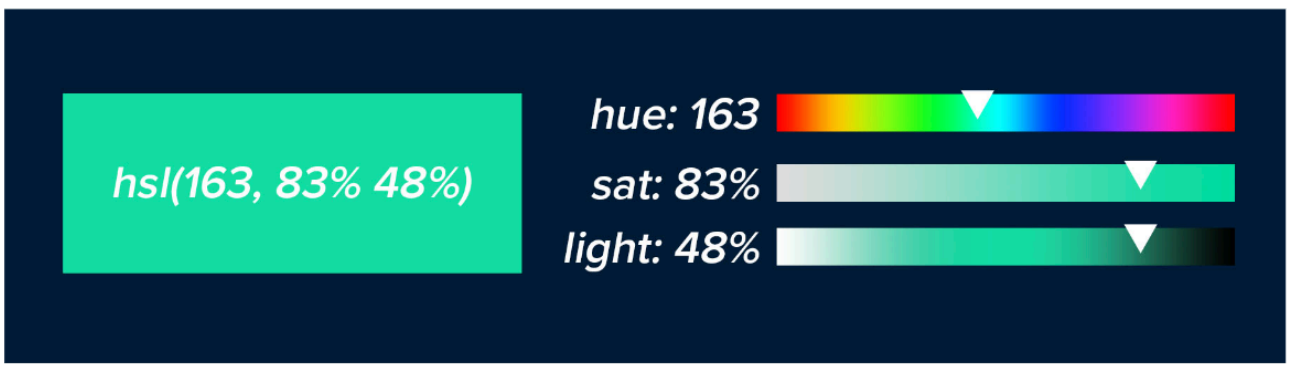
4. Choose when to use mixins vs extensions

- **video notes**
    - -> mixins and extensions are similar
    - -> the end result is the same
    - -> and the aim is both to stop the repetition of code
    - -> when you use mixins rather than extensions, or vice versa
- **notes on the text below the video**
    - Mixins vs extensions
        - general about
            - -> mixins look similar to extensions
            - -> not knowing which to use in which context
        - arguments
            - **-> mixins can have arguments and extensions can't**
        - duplicates
            - -> mixins have duplicate rules <- which is more repeated css than the extension classes
            - -> extensions have duplicate selectors
                - extensions link back to other parts of the code -> so there can be multiple rounds of scrolling to find the property of one element
        - the advice
            - -> **the advice is not to use extensions -> to instead prefer mixins**
            - -> extensions remove the order / predictability of the codebase
                - there is a cost to the reduction in repeated css -> which is this
            - -> this is architecturing css
                - he's suggesting not to write extensions, to use mixins in your own code - but to be able to understand extensions when reading other peoples' code
        - -> **the next chapter is using Sass to alter mixins to a higher level of complexity**
    - Let's recap
        - -> with mixins, you can use arguments but your Sass (aka indented css) will have duplicates in it when it gets compiled into actual css
            - -> arguments don't use arguments
            - -> these duplicate selectors, rather than code
        - -> prefer mixins vs extensions when writing your own code
            - -> but know how to read extensions in other peoples' code

5. Leverage Sass functions to improve mixins

- **video notes**
    - -> variables
        - not making variables for every single thing <- e.g every colour

        - -> you can use Sass to alter existing variables, rather than having to define new ones
- **notes on the text below the video**
    - Conjunction junction, what's your function?
        - -> e.g when you have lots of different shades of green - but the shades of green which you are using are similar
        - -> Sass (indented css) tools for colours
            - -> functions
                - these can take arguments, change them and split out new values
                - -> these functions can work on colours -> darken them, invert, compliment them et al
        - -> to use a Sass function to darken a colour
            - darken
            - arguments are the colour and the amount it's being darkened by
            - example
                - @mixin heading-shadow($colour:$colour-primary, $size: $heading-shadow-size) {
                - text-shadow: $size $size darken($colour, 10%);
                - }
                - -> the shadow of the text is that colour, darker by 10%
                - -> so you don't need to update both -> update the main colour and then the colour of the shadow automatically updates
                - -> this is the case when it complies into css
    - Color voodoo and witchcraft
        - -> **hexcodes can be broken into (r, g, b) coordinates**
        - -> **another way of doing this is hsl (hue, saturation, lightness)**
            - -> hue - ROYGBIV, saturation - how white it is, lightness - how white to black it is

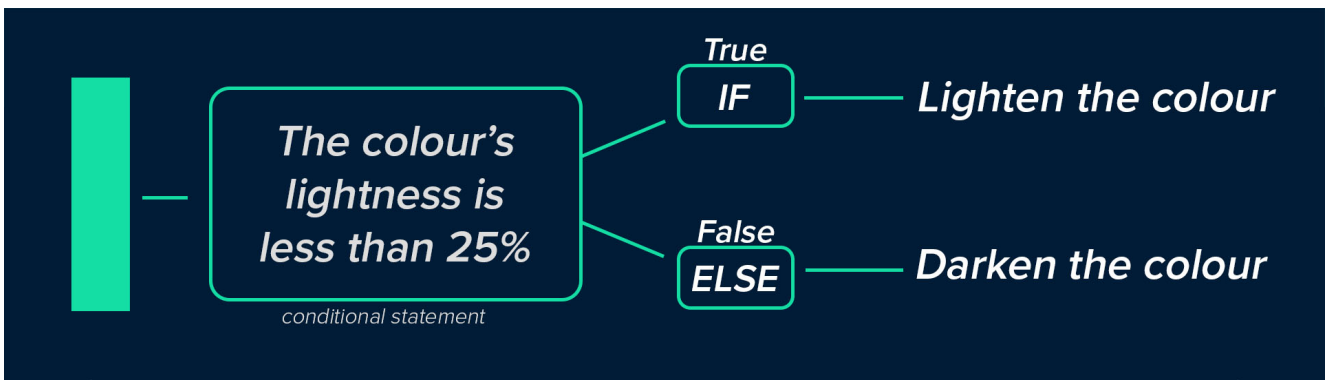


                - -> the colour has a percentage of the last two values, and a degree value of the first (360 on the entire colour wheel)
                - -> using Sass functions on these transforms the colour to another one in the cartesian coordinate system
                    - -> the function darken((in this example is taking a hexcode, transforming it to an hsl value, operating on it, then transferring it back into a hexcode which it returns in the compiled css (when the Sass is converted back into css, Sass is indented css)
                    - -> the function is returning that value
                        - there is a @return keyword
                        - -> if one colour is defined based off of another which is 10% darker than it - it's one less colour to define
        - exercise
            - -> defining a colour as a function of the main one of the page
            - -> the adjust-hue() function - arguments are
                - -> the colour we want to adjust

        - -> then the number of degrees we want to shift the hue by
      - -> he is using it to base one colour off of the primary colour
      - -> then looking a the rendered webpage which comes off of it
    - -> again, the solutions are checked in codepen
    - -> next chapter is making a mixin show different behaviour depending on the context
  - Let's recap!
    - -> functions are blocks of code which can be reused
      - -> Sass comes with functions which can change / manipulate the colours
      - -> there are different channels to do this - r, g, b and hue, saturation and lightness

6. Optimise mixins with conditionals in Sass

- **video notes**
  - the current code is rigid
  - -> functions
  - -> mixins so the values can be updated
  - -> code which reacts depending on the circumstance
    - e.g if the background is dark then make the font light
  - -> there is less code, it's clearer, more concise and there is less of it
- **notes on the text below the video**
  - Truth or dare: using conditionals
    - -> differences in lightness are clearer than differences in darkness
    - -> he's making thresholds for the colours -> if a percentage lightness above 25%, then it gets lightened - and less than that then it gets darker



    - -> if the brightness falls above or below this value, do this to it shadow
      - its an if else statement (conditionals)
        - -> if the colour is light, darken it and visa versa
    - -> @if then the statement <- @if (lightness($colour) < 25%) {...} e.g
      - -> the answer is a boolean
      - -> if the condition is true -> then do this
      - -> this is below (example)
        - @if (lightness($colour) < 25%) {
        - $colour: lighten($colour, 10%);
        - }
        - -> Sass only executes the code inside if the boolean is True
        - -> you can also use else statements - if the condition is false
          - @if (lightness($colour) < 25%) {
          - $colour: lighten($colour, 10%);
          - }@else{
          - $colour: darken($colour, 10%);
          - }
        - -> then they test the code to see if it works when its compiled into css
        - -> you can google the hex codes

- -> exercise
  - -> hovering over buttons <- it's a condition (the mouse is hovering over the element)
  - -> this is changing the hue of the button when its hovered over
    - making it depend on the hue of the buttons's background colour
  - -> the Sass hue function
    - -> there is Sass documentation
    - -> this is to get the hue of the element
  - -> creating a hover mixin with an argument called $color
    - -> then creating a conditional statement -> depending on the background of the button
    - -> hover selectors / mixins
- Just dropped in to see what condition my condition is in
  - -> (lightness($colour) < 25%) <- this is a conditional statement
  - -> there is a comparison operator

Operator	Condition	Result
==	x==y	Returns true if x and y are equal
!=	x!==y	Returns true if x is not equal to y
>	x>y	Returns true if x is greater than y
<	x<y	Returns true if x is less than y
>=	x>=y	Returns true if x is greater than or equal to y
<=	x<=y	Returns true if x is less than or equal to y

  - -> conditional operators
  - -> you can combine two of them in one Sass (indented css) statement
    - and -> e.g @if (lightness($colour) < 25%) and (lightness($colour) > 10% {...}
    - and is a logical operator
    - or -> this is true if any of the conditions nested under it are true (this or this, or this)
      - e.g @if (lightness($colour) < 25%) or (saturation($colour) > 10% {...}
    - -> you don't always need an else statement, e.g
      - @if (saturation($colour) < 50%) {
      - $colour: saturation($colour, 50%);
      - }
      - background-color: $colour;
      - -> don't raise the colour unless the saturation obeys this condition
  - -> exercise
    - -> the hover mixin
      - -> the current one shifts it 30 in one case and 60 in another case
      - ->
    - -> he's using conditional statements around the @if statement, to surround it with other conditions

- -> he's created a range of values of hue using < and > operators in an @if statement
  - if the hue is between these values, change it by this much
  - -> Sass also has a complement() function
  - -> the conditional statements can include any of the symbols in the table above
  - -> pages also have primary / main colours -> $color-primary in this case, which you can base these styles off of
- -> again then checking the entire thing in a codepen / by looking at how the code behaves in a browser
- -> algorithms
  - -> to control program flow
  - -> conditional logic statements
    - the colour of one thing on the page depends on the colour of the rest of them
    - -> rather than manually telling it what the colour is
    - -> the colour of this thing equals the colour is this thing, plus this
    - -> if else, it equals this <- algorithm
  - -> to solve a problem, problem solving tool
  - -> executing instructions in a specific order
- -> the next chapter are putting if/else statements in their own functions

- Let's recap!
  - -> @if / @else statements <- conditionals for Scss (indented css)
  - -> @if statements can be used without @out

7. Create and use Sass Functions

- **video notes**
  - -> if / else statements
  - -> functions
- **notes on the text below the video**
  - Using functions
    - -> he's introduced the concept of a function using a set of very long winded mixins
      - -> mixins are like variables with multiple lines of css (Sass)
      - -> if the argument to one of those doesn't have a default value, in this example it's a boolean conditional statement - he suggests to make a function to give that output
      - -> and then use the output of the function as the unput of a mixin
      - -> rather than having conditional statements in the arguments of mixins
  - Making functions
    - -> the concept of a function <- r, g, b is a function
    - -> he's converting a mixin into a function
    - -> **Sass function syntax**
      - **@function lightness-shift($colour){** <- the function name
      - **@if (lightness($colour) < 25%) {**
      - **@return lighten($colour, 10%);** <- under this condition, return this
      - **}@else{**
      - **@return darken($colour, 10%);** <- under this condition, return this
      - **}**
      - **}**
        - -> in this case, it's the colours being reutrned
    - -> using the functions as input to a mixin
      - *@mixin heading-shadow($colour: lightness-shift($colour-primary), $size: $heading-shadow-size){* <- lightness-shift is the name of the function we just refined, it's returning the colour which the webpage is made of, just darker or lighter and storing that in the colour argument
      - *@if (lightness($colour) < 25%) {*

- *$colour: lighten($colour, 10%);* <- then using the colour argument in the function
- *}@else{*
- *$colour: darken($colour, 10%);*
- *}*
- *text-shadow: $size $size $colour;*
- *}*

- -> exercise
  - -> what we want
    - -> a colour function
    - -> when the button is hovered over, it's background changes
    - -> the colour which is changes is is a pastel version of the original colour
      - a transformation of it
  - -> what we know
    - -> hsl <- s = 100%, l = 90% for pastel colours
    - -> then you get the hue from the hue() function in Sass
  - -> how we get there
    - -> convert the colour which we are basing it off of into an hsl value
    - -> define a function which transforms that into a pastel colour (using what we know) <- we want it to return the pastel colour based off of hte input colour of the webpage
      - -> @function <- syntax
      - -> the input of that function will be the primary colour of the webpage
      - -> then we're defining a function which transforms it into a pastel colour
        - -> hsl
        - -> so, splitting it up into h, s and l values
        - -> then transforming them (according to what we know)
        - -> then combining them into the pastel colour which we return
        - -> then returning that
    - -> then we use the pastel colour returned from the function, as the background colour of the button in its hover state
  - -> how we check that the result makes sense
    - -> we check the result in the browser - in this case it's a codepen
    - -> he then goes back and makes changes
      - -> setting the font colour of the button text equal to the main colour of the webpage - this would depend on a mockup
- -> ja also uses functions

- Let's recap!
  - -> functions <- we like them because they are reusable
  - **Sass syntax with functions**
    - **-> @function <- to define a function**
    - **-> @reutrn <- to return their value**

Quiz: Create efficient, maintainable code with intermediate Sass techniques

- there are 10 questions

Question 1

scss

```scss
.btn{
    color: #0c2461;
    font-size: 2rem;
}
.article{
    color: #0c2461;
    background-color: #fff;
}
```

If you want to refactor the code above, and for this you want to store the color #0c2461 in a variable so that you can reuse it several times in your code, how will you change the code?

- ○ $blue = #0c2461;
- ○ $colour = b;
- ○ pink = $#0c2461;
- ☑ $blue: #0c2461;

- -> refactor -> factors
- -> the factors which the two classes have in common are the colour
- -> we want a variable which stores that defined at the top, and then the name of that variable replaced for the value in each of the lines of code below it
- -> it's a dark blue colour
- -> so it's the first or last one
- -> the syntax is : not =
- -> so it's the last one
- -> check it against open AI and it agrees

- second question
  - -> we have --- Sass / css which is setting the value of a variable
  - -> that variable gives elements which have it a solid border 5px thick
  - -> that hexcode is the same as the blue in the previous question
  - -> then you apply that to a button selector

Question 2

scss

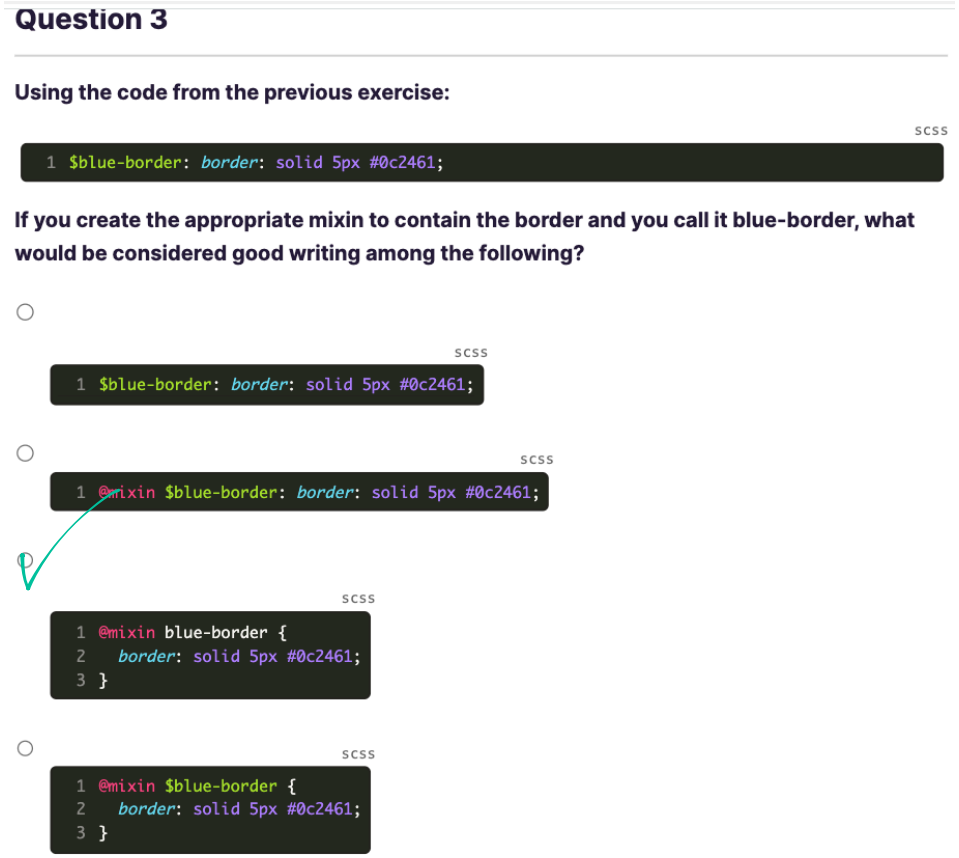
```scss
$blue-border: border: solid 5px #0c2461;
```

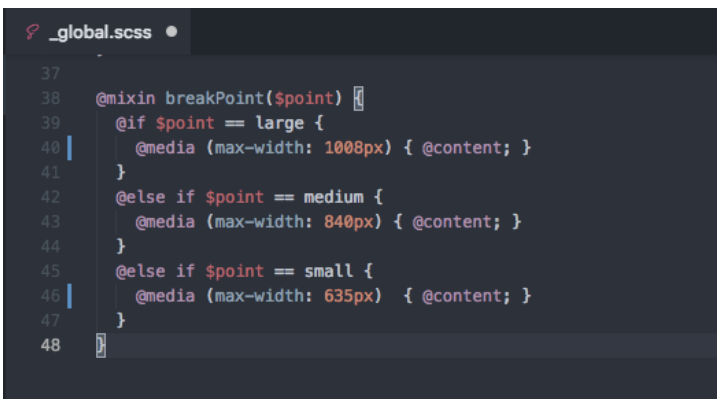
Starting from the code above, what happens if I declare a variable like this to store border and apply it to a .btn selector?

- ☑ A compilation error
- ○ The button will have a blue border
- ○ The button will not have a border

  - -> if we put it into codepen we can see
  - -> the thing is, if you went to apply this to a .button {} class in css -> you can't
  - -> if you do border: $blue-border, then in this case the compiled css would be
    - border: border: solid 5px #0c2461;
    - -> invalid syntax
    - -> you want it to compile border: solid 5px #0c2461;
    - -> so what you do is $blue-border: solid 5px #0c2461;
  - -> either it's the first one or the last one
  - -> it's not that the syntax worked and then wasn't used
  - -> it's that the syntax wasn't written correctly
  - -> so I would say the answer is the first one
  - -> check it against the open AI -> fine

- third question
  - -> right so we know that's the wrong syntax because it's being reused from the previous question in this question
  - -> we also know that's the hexcode for a blue colour
  - -> a mixin -> in other words, mixin syntax
  - -> we're writing a mixin to put the code for the border into it
  - -> the first one isn't a mixin
    - -> so we're left with the other three options
  - -> the second option is wrong because it's reusing the syntax from the code in the question, which is the same as the syntax from the previous question - which we know is wrong
  - -> so it's one of the last two
  - -> the ---
  - -> the difference between those is one uses blue-border and one uses $blue-border
  - -> I would say it's the third option - because $ is what you use for the name of a variable, but here that's referring to the name of a mixin and not the name of a variable
  - -> if you google mixin syntax sass -> you can see it's a blue-border not a $blue-border
  - -> third option
  - -> check it against the open AI
    - -> third. Fine.





- fourth question

Question 4

You have created your blue-border mixin. It is time to use it in your selectors. If you were to use your new mixin in the following class selector:

scss

```scss
.article {
display: flex;
padding: 5px 10px;
color: black;
background-color: $white;
// write your code here
}
```

What form would it take?

- ○ @include $blue-border;
- ○ @mixin blue-border;
- ✓ @include blue-border;
- ○ @extend %blue-border;

- same question as before, just the next part of it
- -> the last option is wrong - that's for extensions not mixins
- -> the right syntax is @include - the second option is wrong
- -> the third option
- -> or not
- -> it's either the first or third options
- -> when you use a mixin,
  - -> in this vase, we're using the value of it -
  - -> we're using the value of it, which is equal to a load of css
  - -> $ refers to value
  - -> so I would go for the first option, and then check it against Open AI
  - -> which says - three - not one
  - -> **when you call a mixin, you use it's entire name, not $it's-name. $it's-name is the syntax for variables**
    - -> so if the question said 'variable', then you would use option 3
    - -> in this case, it's a mixin, so we're using the name of the mixin without the $
    - -> **$is for variables, not mixins**
    - **-> it's like when you call a function, you don't use the value of the function, you just call the function**
    - **-> we're calling the mixin, not using it's value**
    - **-> it can't have a single value, because it's a load of lines of css (by definition) - what is it's value?**
    - **-> the answer is, the mixin doesn't have a value. It's a mixin, a load of lines of css, with no singular value -> if it had a singular value, it would be a variable not a mixin, in which case you would refer to it by it's $value**

- fifth question

Question 5

What is the difference between mixins and extensions?

- ○ There is no difference
- ○ Mixins duplicate selectors and extensions duplicate rules
- ✓ Mixins duplicate rules and extensions duplicate selectors

- this one was made quite explicitly clear in the course - when the Sass mixin compiles into css, it creates duplicate rules
- -> then when the Sass extension compiles into css, it creates duplicate classes / selectors - difference between a class and selector?
- -> check it against the open AI
  - we're fine

- -> asking Open AI about the difference between a class and a selector
  - -> html, css concept
  - -> html <- a class for elements
    - -> <name class="">
    - -> one element can have many of them
    - -> one class can be applied to multiple html elements
    - -> .class <- css syntax for it
  - -> selectors in css
    - -> these can target ids, classes
    - -> element selectors
    - -> **a selector in css is anything which selects html -> it can be a class, an inline element, a class - these are all things which fall under the wider unmbrella of class**

- sixth question

Question 6

If you had to name a placeholder to replace the .content-color class selector, how would you do it?

- ○ $content-colour
- ✓ %content-colour
- ○ .content-colour
- ○ &content-colour

- -> the first thing which comes into mind with 'placeholder' is the text for a search bar
- -> before you click on the search bar, there is text there, and then it goes away when you click on it -> that's placeholder text
- -> a placeholder class is the class which 'goes away' when it's not being used
- -> this was defined with %'s - second option
- -> check it against the OpenAI. Fine.

- seventh question

Question 7

scss

```scss
.content-colour {
  color: blue;
  font-size: 2rem;
}
h1 {
  @extend .content-colour;
}
textarea {
  @extend .content-colour;
}
```

Using the code above as a starting point, how will you compile your code?

Using the code above as a starting point, how will you compile your code?

○
```scss
textarea {
  color: blue;
  font-size: 2rem;
}
```

○
```scss
.content-colour {
  color: blue;
  font-size: 2rem;
}
```

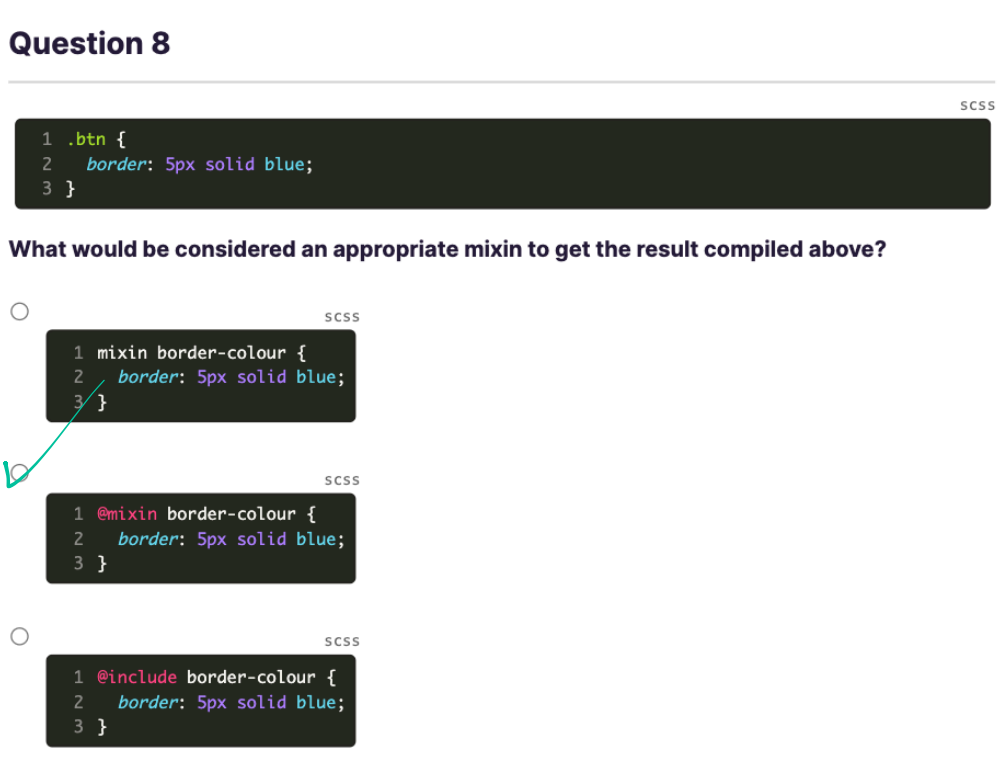
○
```scss
.content-colour {
  color: blue;
  font-size: 2rem;
}

h1 {
  color: blue;
  font-size: 2rem;
}

textarea {
  color: blue;
  font-size: 2rem;
}
```

●
```scss
.content-colour, h1, textarea {
  color: blue;
  font-size: 2rem;
}
```

- the code at the top has extensions in it -> that's Sass
- -> all of the options - they're css
- -> we're going from Sass which is more complicated css, to css which is less complicated
- -> it's a translation exercise
- -> we're trying to translate the Sass into css -> and the question is, what is the final translation?
- -> what is our starting Sass doing?
  - we have a class which is being defined -> that class is called content-color <- it's messing with the content making it blue and the font size larger
- -> what about the classes coming off of it - extending from it?
- -> it's applying those styles to those elements of the webpage
  - -> so when we translate it to css, it starts to list them
  - -> it's not repeating the styles when it's compiled into css like it would with a mixin, it's listing them

- -> so when we go into css, the first two are wrong because they're to generic
  - -> they're missing out nuance you would find in the Sass
- -> then the last two options
  - -> it's the last one
  - -> because - the third one is repeating the styles like a mixin would, and the last one is listing them like an extension would
  - -> this is why he kept on mentioning to write the code with mixins and not extensions -> because when you use extensions, there is less code - which increases the specificity and makes the behaviour of the page more unpredictable
  - -> would go with option 4
- -> check it against open AI
  - -> it's disagreeing with me - option one
  - -> no it's not. Put it in a second time and now it's agreeing with me -> make sure you put it in right when you check it

- question eight



Question 8

scss

```scss
.btn {
  border: 5px solid blue;
}
```

What would be considered an appropriate mixin to get the result compiled above?

scss

```scss
mixin border-colour {
  border: 5px solid blue;
}
```

scss

```scss
@mixin border-colour {
  border: 5px solid blue;
}
```

scss

```scss
@include border-colour {
  border: 5px solid blue;
}
```

    - so, we're looking at a css class
    - it doesn't look like the syntax is wrong -> apply that to an html element and it puts a 5px blue order around it
    - -> this is the same question as the previous one, but backwards
    - -> the previous one was - how do we go from Sass to css? This ons is how do we get from this css to Sass?
    - -> we're getting it into a mixin -> don't see why they don't just use a variable?
    - -> basically, it's saying, we have this cssc class, how do we translate it into a Sass mixin?
    - -> you define the mixin, and then you call the mixin
    - -> the last option is wrong, because @include is what you use to call the mixin rather than define it
    - -> the first option is wrong, because it's missing an @
    - -> would go with the middle option, then check it on the open AI
    - -> it's agreeing with us. Fine.
    - -> why don't we ask it our question about why translate it into a mixin and not a variable?
        - it's saying it's context dependent -> depending on how many times you want to reuse it
        - -> that the variables would be used less widely than the mixins - ?

- question nine
  - -> we have an entire block of indented css, aka Sass

Question 9

scss

```scss
$white: #fff;
$pastel-blue: #0097e6;
$dark-blue: #192a56;

@mixin border-colour($colour){
  @if ( lightness($colour) > 25% ) {
      $colour: darken($colour, 30%);
  } @else {
      $colour: lighten($colour, 30%);
  }

  border: 5px solid $colour;
}

.article {
  display: flex;
  padding: 5px 10px;
  color: black;
  background-color: $white;

  @include border-colour($dark-blue);
}
```

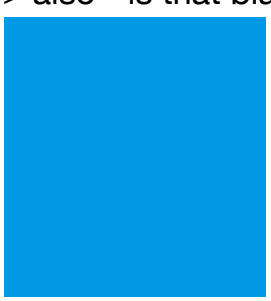
html

```html
<div class="article">Read me!</div>
```

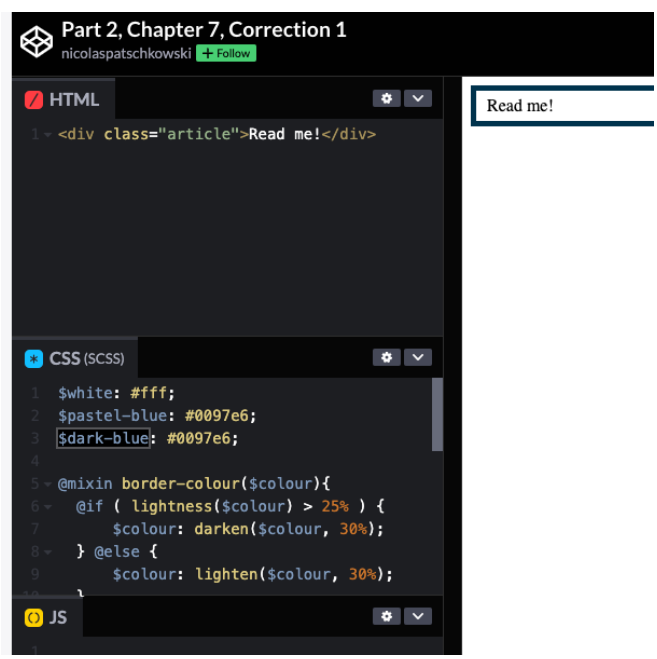
Using the code above as a starting point, what color will the border of .article be if you replace $dark-blue with $pastel-blue?

- ○ White
- ○ Black
- ○ Dark blue
- ○ Light blue

  - -> that's being used on an html div
  - -> what I would do is just put it into a codepen -> does codepen use Sass or just css? Try it.
  - -> or VSCode, but it might be more tedious -> codepen it is
  - -> can't get it to use Sass, so would go to one of their exercise files which uses Sass and then paste it in there
  - -> so the experiment we're running is -> replacing dark blue with pastel-blue, then the answer is whatever happens to the border colour of read-me
  - -> this is the code pen -> it's black
  - -> would put it into the open AI and see if it agrees
    - -> also - is that black or a very dark blue?



    - <- this is the 'dark blue' -> we have black

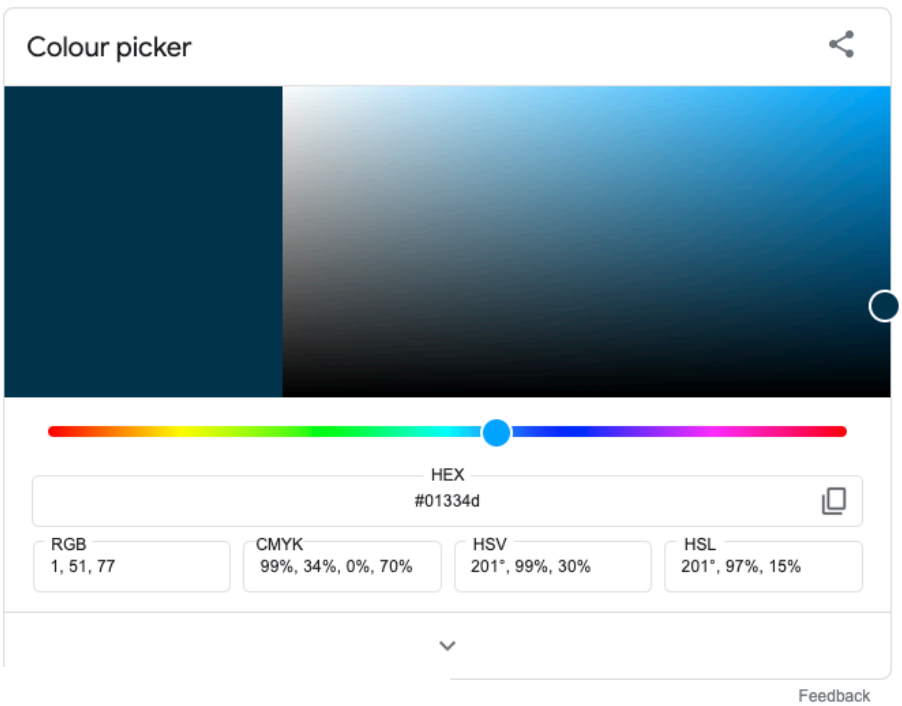


  - the open AI is disagreeing with us and saying light blue
  - -> do it again in another codepen and see
    - black
    - -> I'm disagreeing with the Open AI
  - -> you can also - try putting it into the Open AI again

- -> now it's spitting out a different answer -> which is dark blue
- -> so is what we have in the codepen a black or an ultra dark blue?

- #01334D <- this is the hexcode which is coming out of the codepen
- -> black or bark blue?!

- -> think that's a dark blue -> black would be in the bottom right not just above it
- -> would go with option three, then throw it into a new tab Open AI a third time to see what it comes out with this time
- -> this time it's coming out with a light blue
- -> it never came out with black
- -> so I would go with option three,



it's a dark blue

Question 10

```scss
$white: #fff;
$pastel-blue: #0097e6;
$text-color: #192a56;

@function background($text-color) {
  @if ($text-color == $white) {
    @return $pastel-blue;
  } @else {
    @return $white;
  }
}

.btn {
  color: $text-color;
  background-color: background($text-color);
}
```

Using the code above as a starting point, what would be the background-color of .btn?

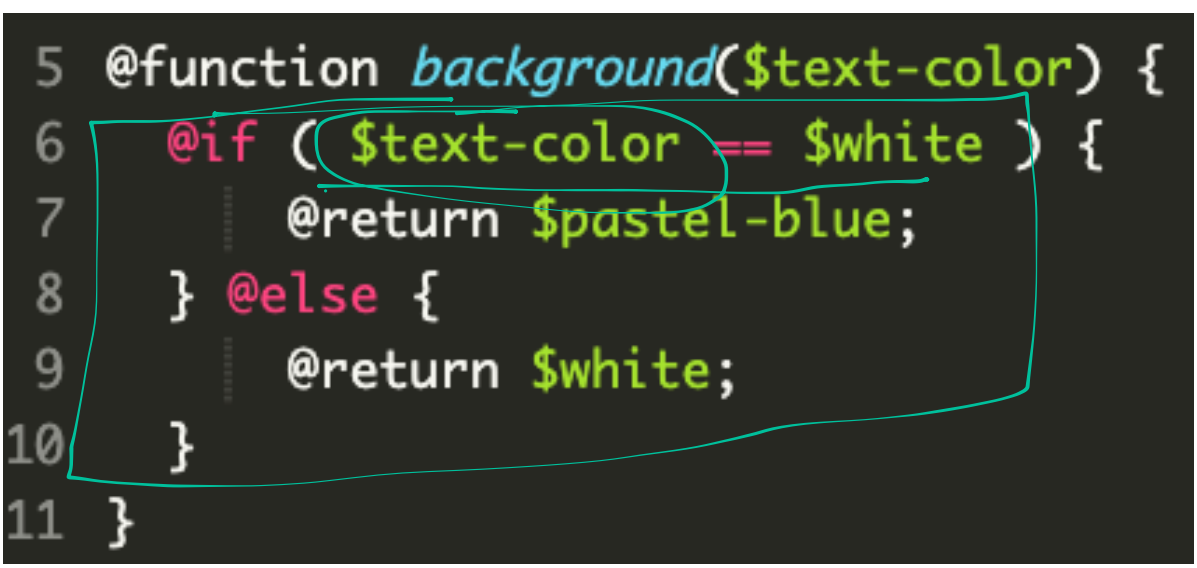
- ○ None, there will be a syntax error
- ○ #0097e6
- ✓ #fff
- ○ #192a56

- question ten
  - -> so we have some Sass
  - -> put it into a codepen, then see
  - -> inspect the colour carefully
  - -> and then check it against what the OpenAI says

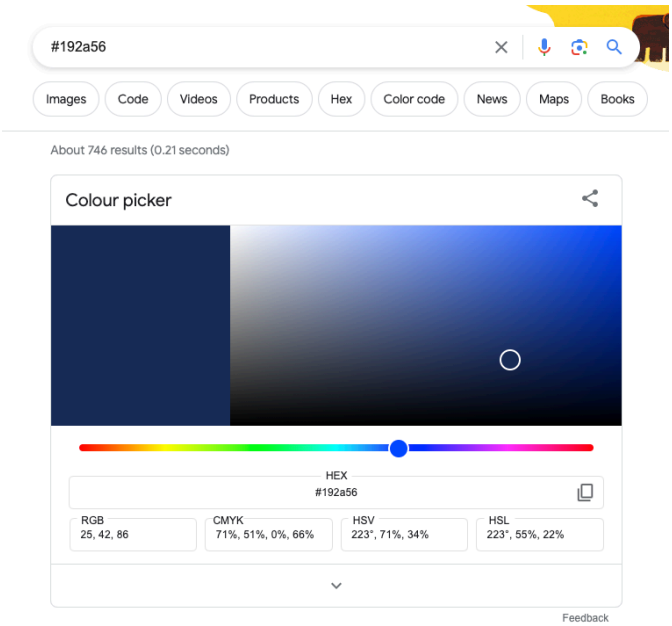
- in a codepen
  - -> I genuinely don't know, because they haven't given us any html to work with - just a load of Sass?
  - -> ?!
  - -> I would go with the - this is the thing, even if you reuse the html from the previous question (in a codepen not IDE), then the appearance of the thing is staying the same
  - -> is it even a button, or just a div?
- -> I would go with the first option -> none
- -> the OpenAI says-
  - -> it's saying two
- -> the question says what "would be" -> implying that the button doesn't exist
- -> say it did, then what colour would it be?
  - -> and we don't care about the colour of the text, just the background

- looking at it again

  - the background colour is "background($text-color)"
  - the first thing is -> what is "background($text-color)"?
    - -> understand that and then you have the answer



- the first thing is, does text-color = white?



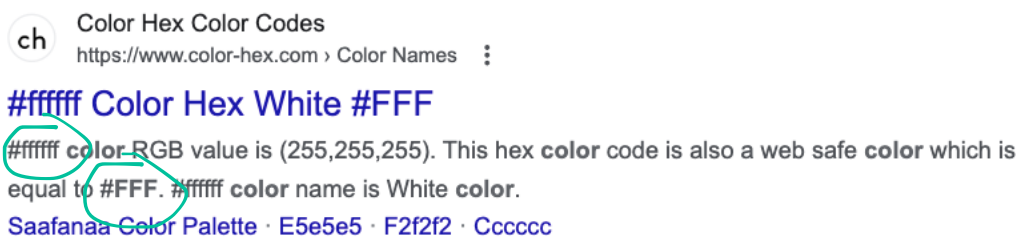

```
1 $white: #fff;
2 $pastel-blue: #0097e6;
3 $text-color: #192a56;
4
```

- no, so this is the value which "background($text-color)" takes ->
  - which is, what exactly?
  - -> the value stored in $white
  - -> which is

```
} @else {
    @return $white;
```

```
$white: #fff;
```

  - -> which is white
- -> the one question I have is, is #fff an actual colour?



  - whenever lowercase f's are used -> in this case there are - six of them
  - -> and then when there are uppercase f's there are three of them
  - -> we have three lowercase f's
  - -> will that break the code?
    - the OpenAI is saying "Yes, #fff is a valid color representation in Sass. "
  - -> I would go for white then
  - -> and reject its answer of light blue
  - -> I would go with the third answer and reject its answer of the second one
  - -> put it in again to see what is comes out with
  - -> it's saying two but I would disagree and go with three
  - -> they're all right. Fine.