Slidecrafting with Quarto

Emil Hvitfeldt

2024-08-18

Table of contents

Preface

Hello! This book is about what I like to call **slidecrafting**; The art of putting together slides that are functional and aesthetically pleasing. I will be using quarto presentations throughout the whole book. Some of the advice will transcend quarto presentations and apply to all kinds of slide technologies, but the book is written with quarto in mind.

I think of slidecrafting as an art because of the inexact nature of many of the decisions you will be making. After all many slide decks can be distilled down into a series of still images, each of which should be carefully crafted.

This book will not be able to teach you everything you need, but it will cover the parts of the process that stay constant from deck to deck, so you can focus on the content and develop your brand/style.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

Part I

Theming

2 10 minute theme

This chapter will show you how to set up a fully functional theme in about 10 minutes. The remaining chapters of this section go into much more detail, showcasing tips and tricks along the way. >

3 Colors

3.1 Finding Colors

When finding your colors you will mainly look for 3 different types of colors:

- background colors,
- text colors,
- contrast colors

this is a simplistic model and it will do us just fine. In general, I think having 3-6 colors is about the right amount of colors. This doesn't mean that you couldn't have 10 colors in your theme, but that you should be very deliberate in your choices.

For a small color theme, you would want a light and dark color for the background and text color. Dark text on a light background, or light text on dark background. Lastly, you pick a contrast color, something that looks good with both your background and text color.

This is what you get with the default themes that are provided in Quarto, and you can see them here in this gif:

This is also a perfect place to start looking for inspiration. using these themes with one or two modifications might get you all the way where you want to go. It is in general a good idea to look for inspiration in other people's work.

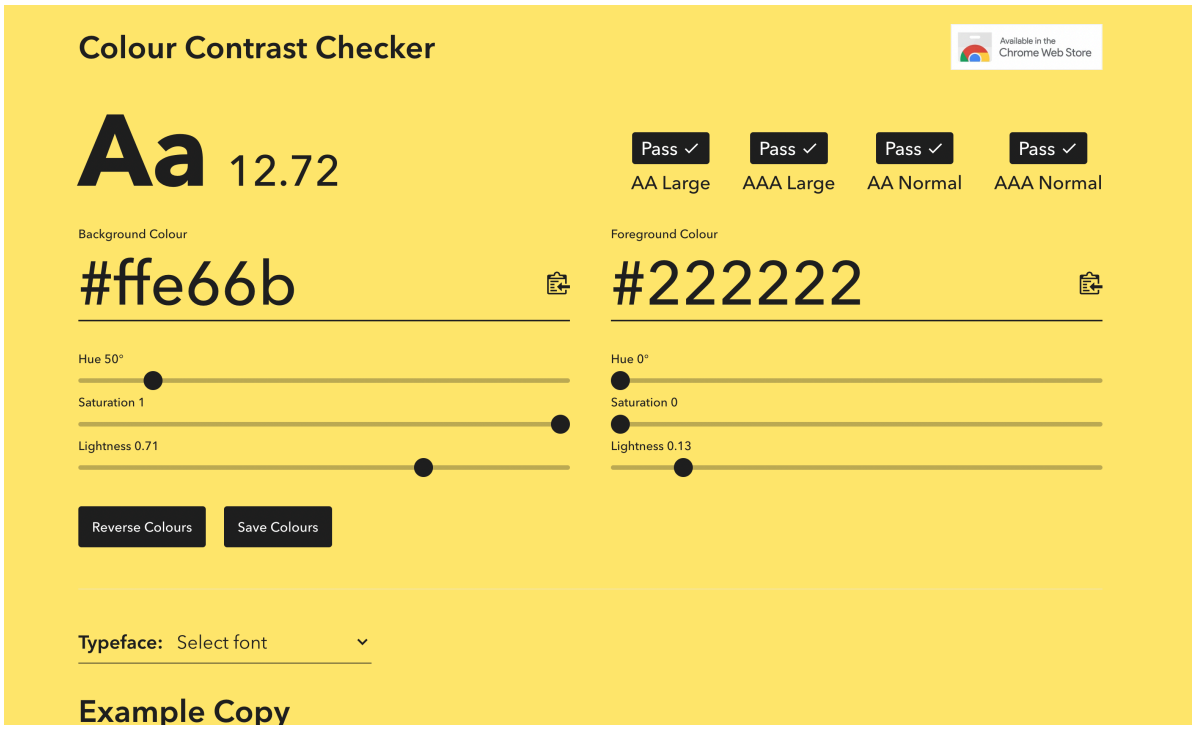
Another personal favorite place of mine to go color theme hunting is on pinterest. I do a google search for "Pinterest color palettes" and go wild.

If you have any specific ideas in mind you can expand your search to include words like "sea", "beach", "Halloween", or "pastel".

The main thing you need to keep in mind, and the biggest difference from other types of colors you may have worked with, such as in data visualization, is that you need to have **high contrast** between your colors. This is by far the most important thing that separates a good theme from a bad theme. The goal for your slides is for other people to see them, if your contrast is low then people can't.

There are many color contrast checking websites out there, I like the https://colourcontrast.cc/ and Color Contrast Checker by Coolors. If possible I try to have a contrast of at least 12, but

something like 10 will be okay from time to time. Which is quite a high contrast without being impossible to hit.



This contrast requirement means that both your background and text color will be quite dark and light, as it is quite hard for most highly saturated colors to have high contrasts to anything else.

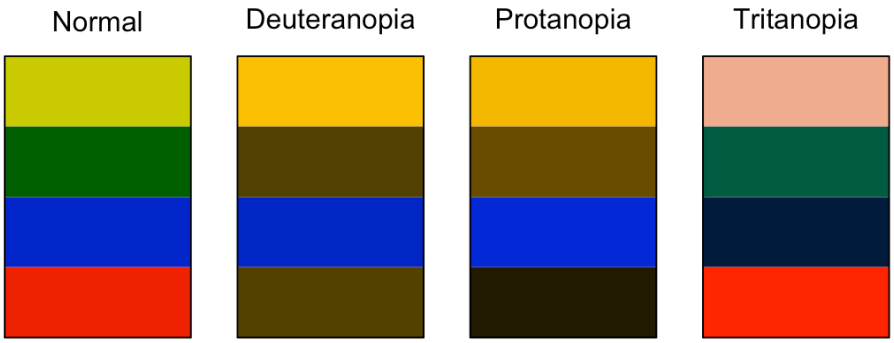
> **Warning**
>
> You should try to avoid pure black and pure white. These colors can be a bit much and can be unpleasant to look at for long periods of time.

This contrast is related to font size, the smaller and finer the text is, the more important it is that you have good contrast.

Next, we take a look at **highlight** colors! This is my term for the colors you use to add some pop and to direct the viewers' eyes. These colors are used for anything from link colors, highlighting, buttons, and artistic elements.

you generally want 1 to 3 of these colors. Having at least 1 is perfectly sufficient and you can use it to great effect to direct the colors. 3 colors are where I'm still comfortable that they don't get diluted. Using too many highlighting colors can confuse your viewers.

These colors should be different enough from the background and text color that they stand out. If you are using multiple highlighting colors you should make sure that they are colorblind-friendly with each other. I like to use the `check_color_blindness()` function from the prismatic package.



As we see above, the green and red colors don't work well together because they are almost identical for people with Deuteranopia.

To recap:

- I think of or search for a palette I like
- I pull out 1-2 background colors, 1-2 text colors, and 1-3 highlight colors
- I use my color contrast checkers to validate and possibly modify my colors so that they are within range
- I check that the colors I have are colorblind-friendly
- ...
- Done!

As long as I can keep the searching under 10 minutes the whole theme creation doesn't take more than 15 minutes.

3.2 Applying Colors

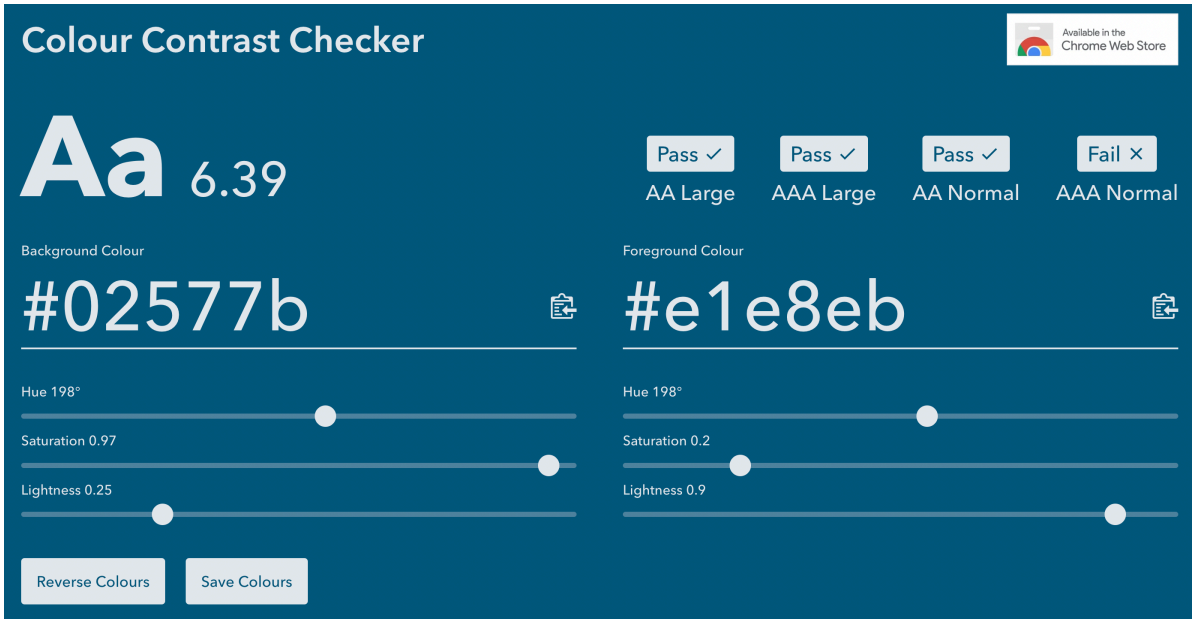
Let us try all of that in practice. I found this nice blue and yellow color palette on Pinterest.

using a color picking tool, I love ColorSlurp I can extract the colors to be

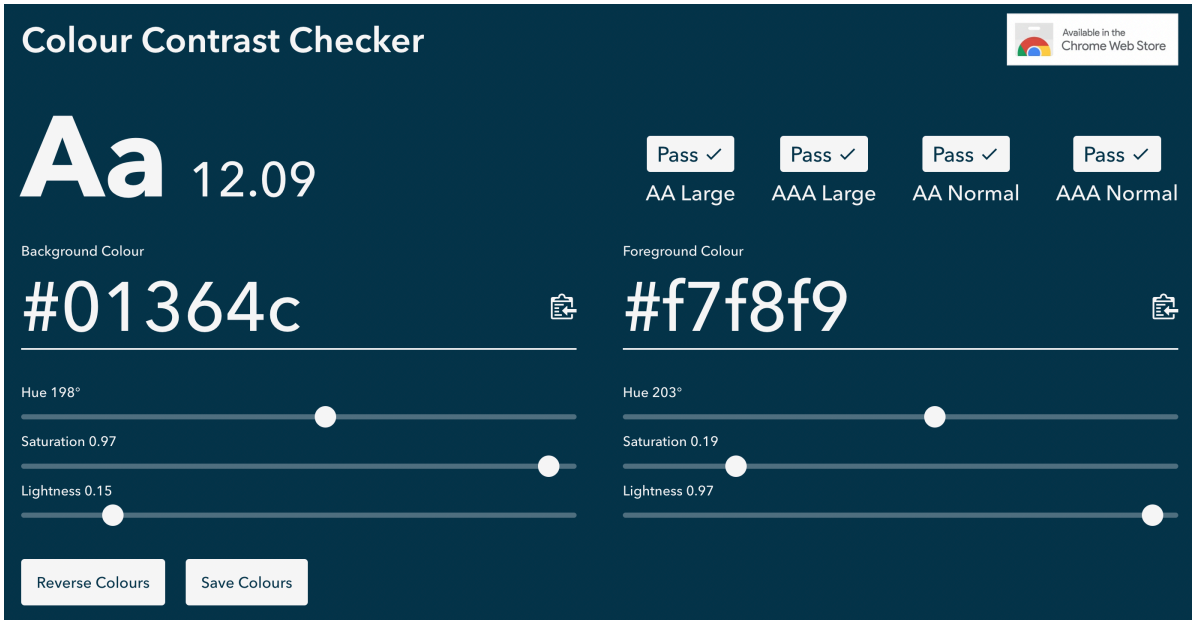
```
*Orient*
02577B

*Fountain Blue*
5CB4C2
```

```
*Morning Glory*
99D9DD

*Mystic*
E1E8EB

*Selective Yellow*
F4BA02
```

I'm thinking I want to use dark blue as my background colors, and the lightest color as my text color. Before I do any modification I get the following



And by playing the sliders a little bit I have a contrast and some colors I'm happy with



We now open up our `.scss` file and fill in a couple of values. Many of the colorings are done by relations, so we can get a lot done by setting `$body-bg`, `$body-color`, and `$link-color`. This needs to be done inside `scss:defaults`.

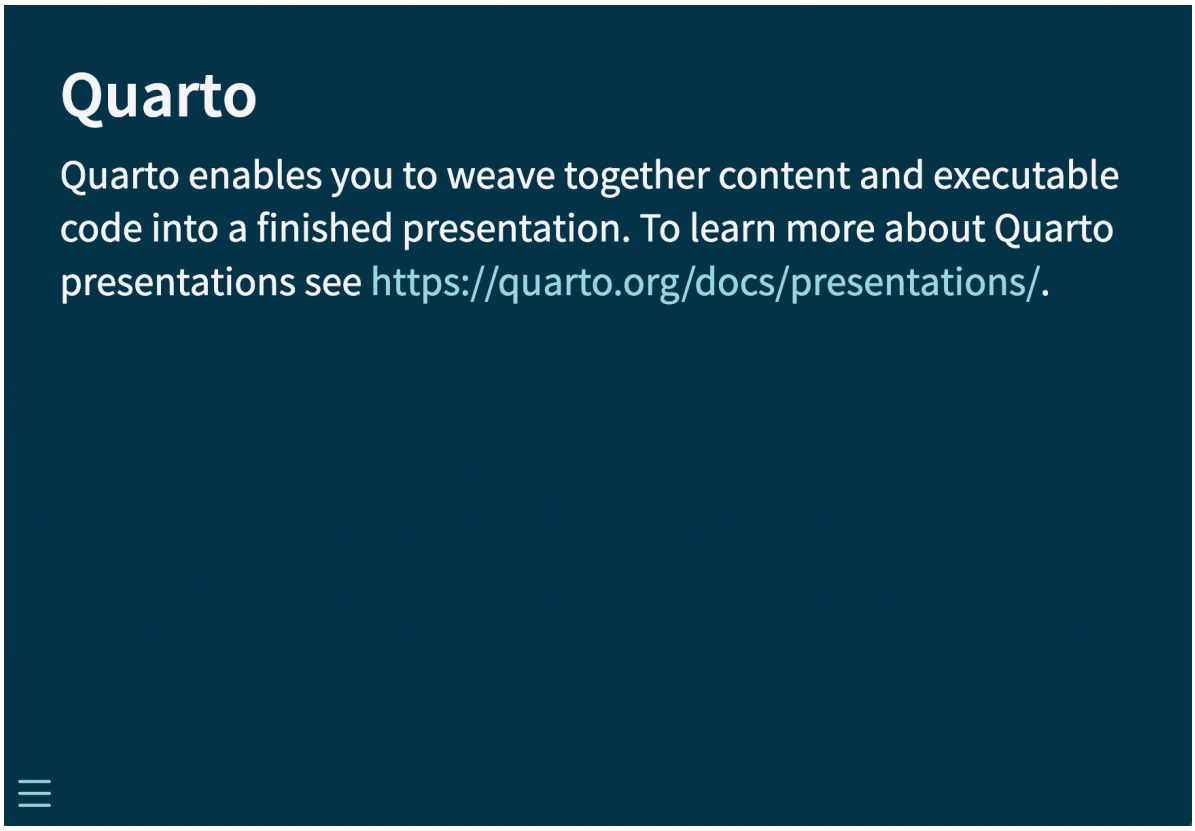
```
/*-- scss:defaults --*/
$body-bg: #01364C;
$body-color: #F7F8F9;
$link-color: #99D9DD;

/*-- scss:rules --*/
```

While the above configurations are perfectly fine, I find that using sass variables to be clear, and it helps us tremendously if we start making more changes. So I create variables all with the prefix `theme-` and descriptive names so I know what is what.

```
/*-- scss:defaults --*/
$theme-darkblue: #01364C;
$theme-blue: #99D9DD;
$theme-white: #F7F8F9;
$theme-yellow: #F4BA02;

$body-bg: $theme-darkblue;
$body-color: $theme-white;
$link-color: $theme-blue;
```

```
/*-- scss:rules --*/
```

This is more code, but now I can read at a glance what is happening. This gives us the following colors on our slides. All done with minimal code and effort. Using one of the highlight colors here to color the links, which also affects the hamburger menu and the progress bar at the bottom.



There are several sass variables that are used to control how our slides look. Notice how many of the values are defined as transformations of other values. So by setting `$body-bg`, `$body-color`, and `$link-color` we automatically gets things like `$text-muted`, `$selection-bg`, `$border-color` with values that works pretty well.

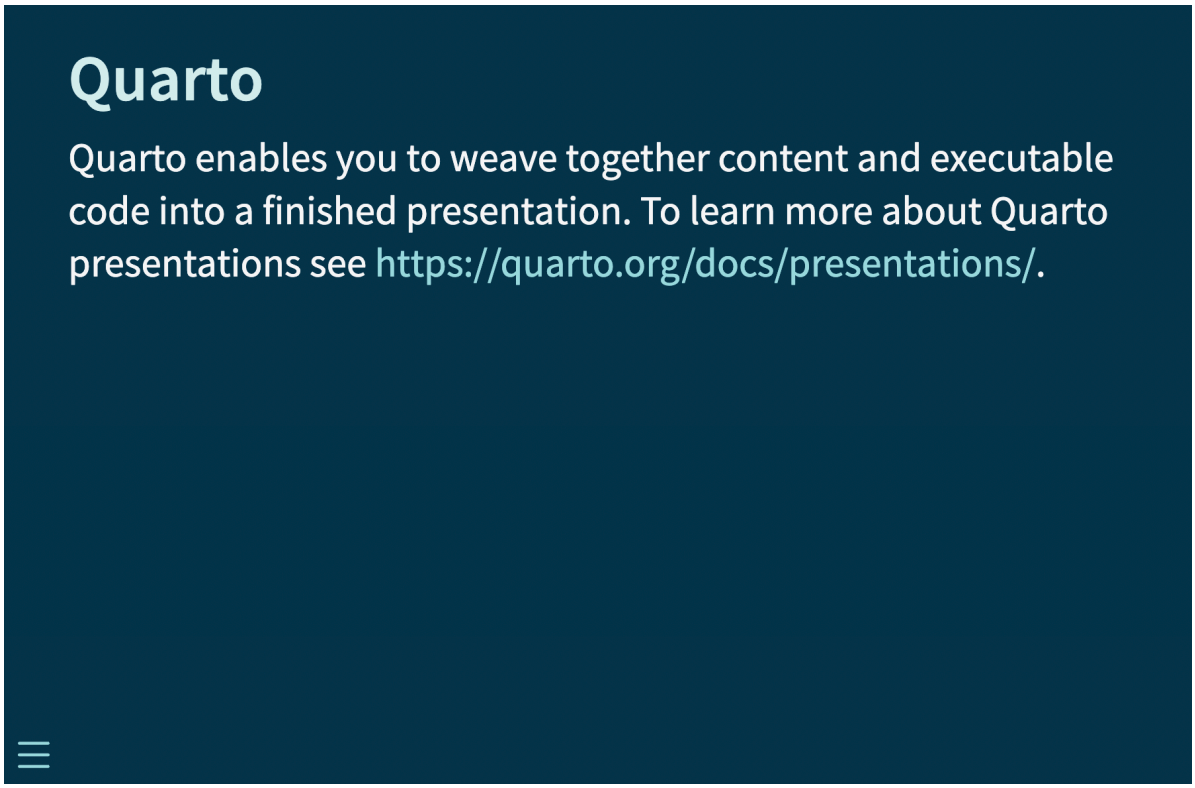
Let us modify our theme just a bit more before moving on to fonts. We can use sass color functions to modify colors based on our theme.

I want the headers to pop a little bit more, So I'm going to see if I can make them ever so slightly lighter blue. I see that the sass variable that controls the header color is `$presentation-heading-color` and that it defaults to `$body-color`. I use the `lighten()` function with `$theme-blue`, iterating a couple of times to find the perfect value.

```scss
/*-- scss:defaults --*/
$theme-darkblue: #01364C;
$theme-blue: #99D9DD;
$theme-white: #F7F8F9;
$theme-yellow: #F4BA02;

$body-bg: $theme-darkblue;
$body-color: $theme-white;
$link-color: $theme-blue;
$presentation-heading-color: lighten($theme-blue, 15%);

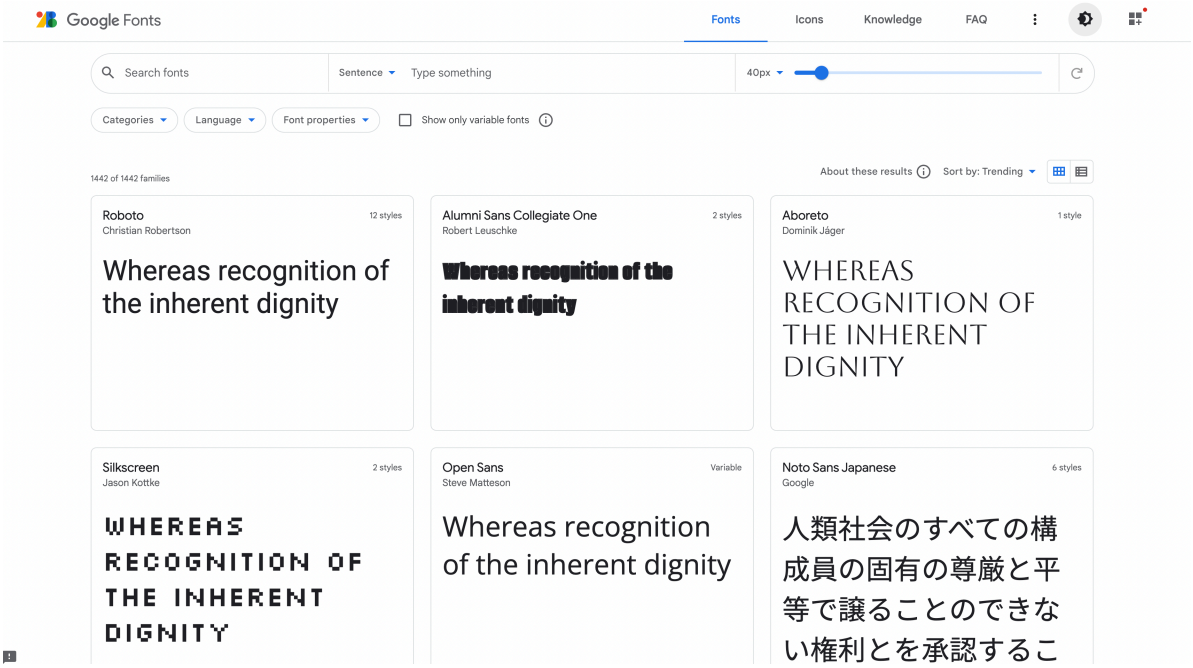
/*-- scss:rules --*/
```



4 Fonts

4.1 Finding Fonts

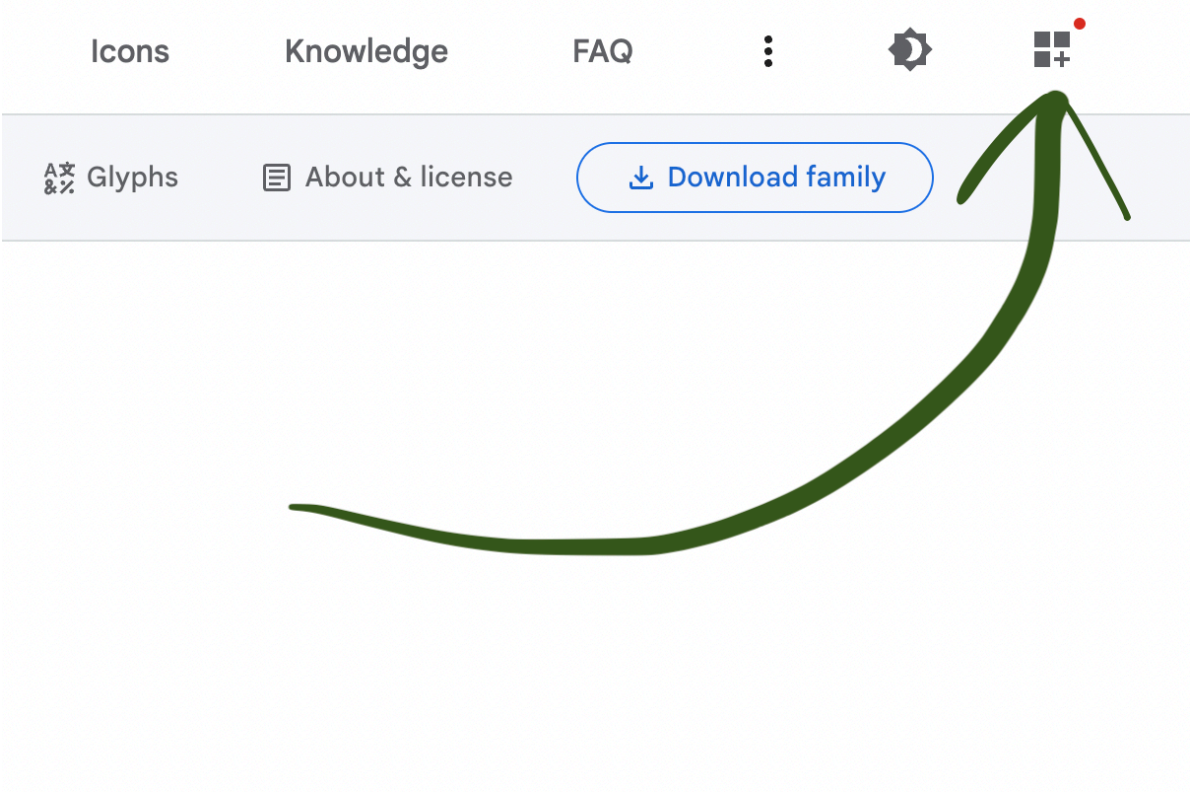
We find fonts the same way we find color; using our favorites of lots of googling. I always gravitate towards fonts.google.com. Generally, it is nice to use these online fonts because they are free and you don't have to embed/ship them if you want to share your slides with others.



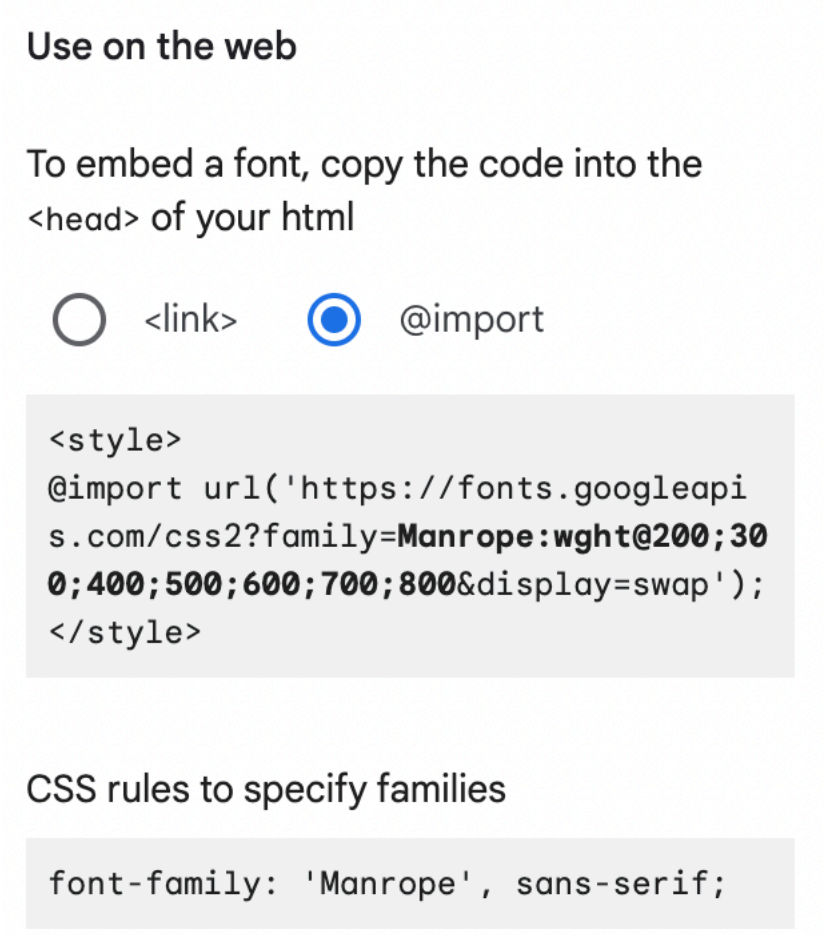
Once we are in here to can search around, looking for a font you like. For these slides, I'm going with Manrope for the text, and IBM Plex Serif for the headers. You must find a legible font, with a couple of styles and bold/italics support. This is going to make your life a lot easier once you get going.

To use "select" these fonts for use, you click on these links for each font type combination.

Then you click this button to have a sidebar menu pop up.



This menu lets you select and deselect the fonts you have selected. When you are done, you can go to the "Use on the web" section, and click `@import`.



And you want to copy the code inside the `<style>` tags. We are now ready to apply these fonts to our slides!

4.2 Applying Fonts

Start by adding the `@import` calls we found in the previous section. This should again go into the `scss:defaults` section of the `.scss` file. to modify the font we have 2 sass variables. First, we have `$font-family-sans-serif` to modify the general text, and `$presentation-heading-font` to modify the headers. Applying these changes gives us the following `.scss` file

```
/*-- scss:defaults --*/
$theme-darkblue: #01364C;
$theme-blue: #99D9DD;
```

```
$theme-white: #F7F8F9;
$theme-yellow: #F4BA02;

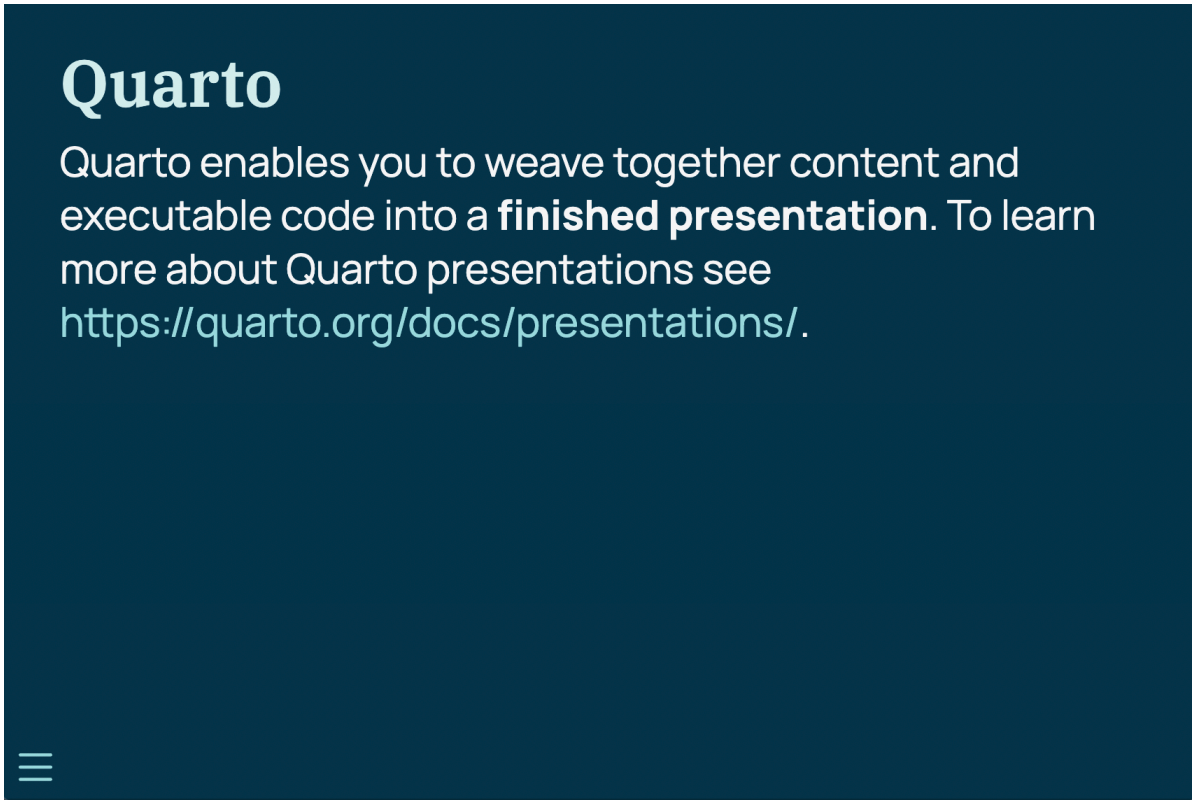
@import url('https://fonts.googleapis.com/css2?family=IBM+Plex+Serif:ital,wght@0,100;0,200;0
@import url('https://fonts.googleapis.com/css2?family=Manrope:wght@200;300;400;500;600;700;8

$body-bg: $theme-darkblue;
$body-color: $theme-white;
$link-color: $theme-blue;
$presentation-heading-color: lighten($theme-blue, 15%);

$font-family-sans-serif: 'Manrope', sans-serif;
$presentation-heading-font: 'IBM Plex Serif', serif;

/*-- scss:rules --*/
```

Which when renders results in the following slides.



Notice how the fonts we used allows us to bold the words “finished presentation”.

Another thing you sometimes need to change depending on the font is the sizing of the text. The sass variable `$presentation-font-size-root` controls this, and defaults to `40px`. Changing this one variable will affect everything on your slides.

You can also change the code font, this should ideally be a monospaced font. This is done using the `$font-family-monospace` sass variable.

5 Theme

This way you can have black/white variants, white/red/blue, or any number of different styles. If you are familiar with xaringan you have seen an `inverse` theme variant already.

5.1 What is a variant?

When we are slidecrafting, and you start having many slides. It can be helpful to bucket them into fewer types of slides. This way you can reuse the same style many times with minimal copy-pasting.

Using colors to create multiple variants of the same theme allows us to quickly add similar looking, yet different styles. The `inverse` theme variant of {xaringan} was a dark grey background slide, that accompanied the white background themed default. I and many other people found this `inverse` theme helpful for creating a break in slides. Typically using it as a section break or a background on which to show a quote.

You can also imagine having a couple of more similar theme variants that are used to denote theory/practice, idea/execution, pros/cons. The opportunities are endless, and we are not limited to only 2. I have in the past used themes that slowly changes colors as the slides progressed through the topics.

5.2 The SCSS basics

Fortunately, adding this behavior in quarto revealjs slides. We need 2 things:

1. Mark slides that have each theme variant
2. include css/scss to style each theme

In our `.qmd` document we can denote each slide as a class with the following syntax

```
## Slide

## Slide {.variant-one}

## Slide {.variant-two}
```

this gives the slides the corresponding `css` class which we can create styles for. Notice how the first slide doesn't have a variant specified. Depending on your usage, it is easier to have a good base style, and only use `{.class}` to specify when you want a different class.

create a `.scss` file and add it to the themes in the yaml section

```
format:
  revealjs:
    theme: [default, styles.scss]
```

And in the `.scss` file, we add the boilerplate information.

```
/*-- scss:defaults --*/

/*-- scss:rules --*/
```

Under the `/*-- scss:rules --*/` section we can now specify all the css rules we want. And we do this by prefixing `.variant` to each style. As an example, if we want to change the color of the text we use `.variant-one {color: blue;}`, or the link color `.variant-one a {color: green;}`.

You can quite quickly end up making many changes. And this is where I find it helpful to use scss nesting. Nesting allows us to rewrite

```
.variant-one {
  color: #d6d6d6;
}

.variant-one h1, h2, h3 {
  color: #f3f3f3;
}

.variant-one a {
  color: #00e0e0;
}

.variant-one p code {
  color: #ffd700;
}
```

as

```
.variant-one {
  color: #d6d6d6;
  h1, h2, h3 {
    color: #f3f3f3;
  }

  a {
    color: #00e0e0;
  }

  p code {
    color: #ffd700;
  }
}
```

I find it quite readable and I encourage you to follow the link and read more about it! Using this syntax, having multiple different variants is quite effortless, and many IDEs will help highlight and collapse this type of syntax.

```
.variant-one {
  color: #d6d6d6;
  h1, h2, h3 {
    color: #f3f3f3;
  }

  a {
    color: #00e0e0;
  }

  p code {
    color: #ffd700;
  }
}

.variant-two {
  color: #a6a6d6;
  h1, h2, h3 {
    color: #222222;
  }

  a {
    color: #f22341;
```

```
  }

  p code {
    color: #ff00ff;
  }
}
```

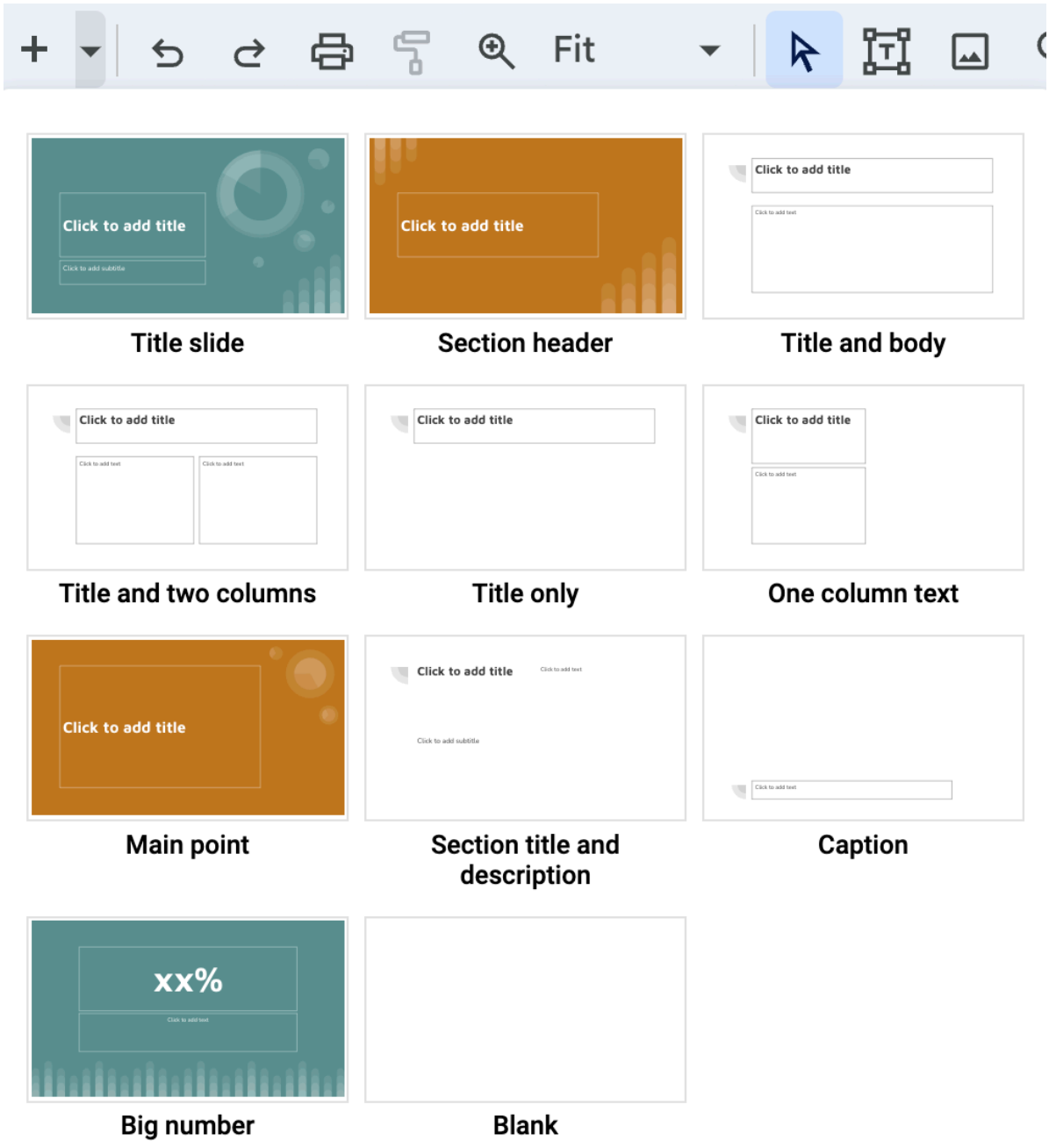
And that is all that is needed! I have taken the liberty to create two themes to show case how this is done in practice:

https://github.com/EmilHvitfeldt/quarto-revealjs-inverse

https://github.com/EmilHvitfeldt/quarto-revealjs-seasons

5.3 What is a slide theme?

The inspiration for this style of slidecraft isn't anything new. If you have used conventional slide-making tools you have seen a dropdown menu before



With these menus, you can swiftly select the style of slide you intend to write and fill in the content. I find that for some presentations that I all I need.

Below is one such theme I created (it is a real revealjs slide deck, touch and advance the slides)

Github Website

Instead of carefully managing the style of all the elements of each slide. They all have an overall slide theme that controls all the content on the slide. This controls colors and sizes but can go further and control backgrounds and even the positioning of elements.

Take `.theme-section1` as an example. Not only are the text and colors modified. The text region is being modified such that the text isn't going to overlap with the globe on the right. Setting this up beforehand is quite nice. While the backgrounds might seem complicated, they are all SVGs, but you can use any other type of image or none at all.

Once you have created the theme, your slides will look like this:

```
## Happy slides {.theme-title1 .center}

## Fancy Section {.theme-section3 .center}

### Less Fancy Subtitle

## Funny title {.theme-slide1}

Content

## Exciting title {.theme-slide2}

Content

## Sad title {.theme-slide3}

Content
```

Each slide will have minimal CSS, just one or two classes specified on the slide level.

5.4 How to create your own

We will assume that we start our project the same way as we did in previous blog posts. What we will be doing is creating several `CSS` classes. I find it easier to prefix all of them with `.theme-` but that is not a requirement. We will also be using the feature that Sass lets us create nesting rules css.

We start with a simple class rule

```
.theme-slide1 {
}
```

if we are following my advice on creating css color palettes then we can use those to specify header colors

```
.theme-slide1 {
  h3 {
    color: $theme-blue; // or #5CB4C2
  }
}
```

And we can specify anything want in here. Note that anything inside `h3` applies to all `h3` headers in `.theme-slide1` slides.

```
.theme-slide1 {
  h3 {
    color: $theme-blue; // or #5CB4C2
    font-size: 2em;
  }
}
```

We could specify specific background colors

```
.theme-slide1 {
  background-color: #E1E8EB
  h3 {
    color: $theme-blue; // or #5CB4C2
    font-size: 2em;
  }
}
```

Or we could specify background images, for reasons I don't want to get into, this is the way to include an image nicely. With `../../../../../assets/slide1.svg` being a valid path to the slides. You may have to modify the number of `../` for this to work

```
.theme-slide1 {
  &:is(.slide-background) {
    background-image: url('../../../../../assets/slide1.svg');
    background-size: cover;
    background-position: center;
```

```
    background-repeat: no-repeat;
  }
  h3 {
    color: $theme-blue; // or #5CB4C2
    font-size: 2em;
  }
}
```

depending on your slides you might have repeated styles a lot. Sass has a way to help us with `@mixin` and `@include`. You can create a `@mixin` with several styles, and then instead of copying around all the styles, you can `@include` the mixin for the same effect. Using this we now have the following

```
@mixin background-full {
  background-size: cover;
  background-position: center;
  background-repeat: no-repeat;
}

.theme-slide1 {
  &:is(.slide-background) {
    background-image: url('../../../../../assets/slide1.svg');
    @include background-full;
  }
  h3 {
    color: $theme-blue; // or #5CB4C2
    font-size: 2em;
  }
}
```

Lastly, if you are using images the way I'm using them, you will need to change the text regions to avoid text overlapping with the background image, we can use the `margin-` rules for that

```
@mixin background-full {
  background-size: cover;
  background-position: center;
  background-repeat: no-repeat;
}

.theme-slide1 {
  &:is(.slide-background) {
```

```
    background-image: url('../../../../../assets/slide1.svg');
    @include background-full;
  }
  h3 {
    color: $theme-blue; // or #5CB4C2
    font-size: 2em;
  }
  h2, h3, h4, h5, p, pre {
    margin-left: 100px;
  }
}
```

I hope you can see that with this style of slidecrafting, the skies the limit. The style sheet for the above example can be found here.

5.5 More Examples

Below is another theme, following very closely the construction of the previous

Github Website Scss file

Another approach to creating these styles of themes is to use Lua to further expand the capabilities of the slides

Github Website Lua file

6 Styling

This chapter is meant to show how we can style specific elements on a quarto slide

6.1 Using CSS Classes

The last tip for this blog post is the idea of using CSS classes, which is a quick and powerful way to add styling to your slides.

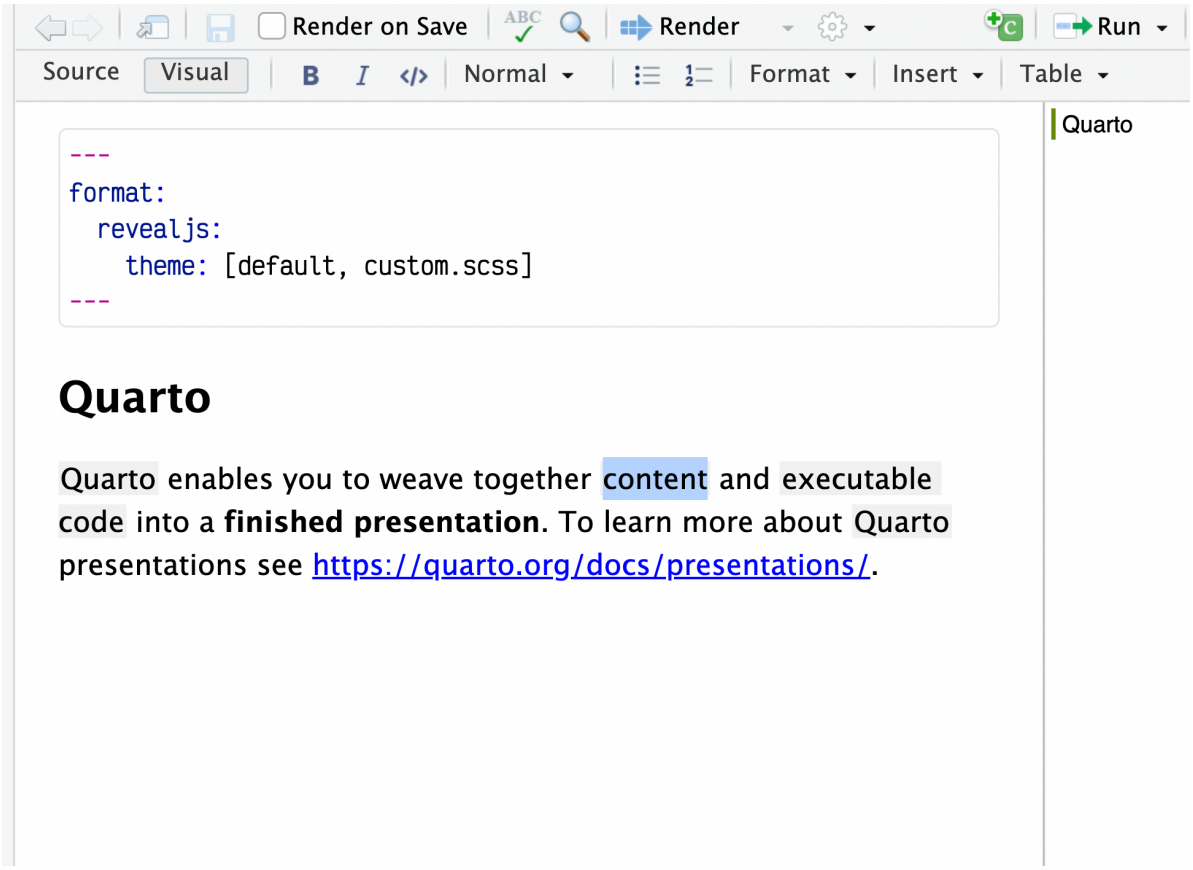
Remember how we have 2 highlighting colors? We should have a way to apply these colors in our slides. For starters let us add a way to turn text these colors. Below are two CSS classes, named `.blue` and `.yellow`, that change the `color` and make the text bold with `font-weight: bold;`. Note that we need to put these classes into the `scss:rules` section.

```
/*-- scss:rules --*/

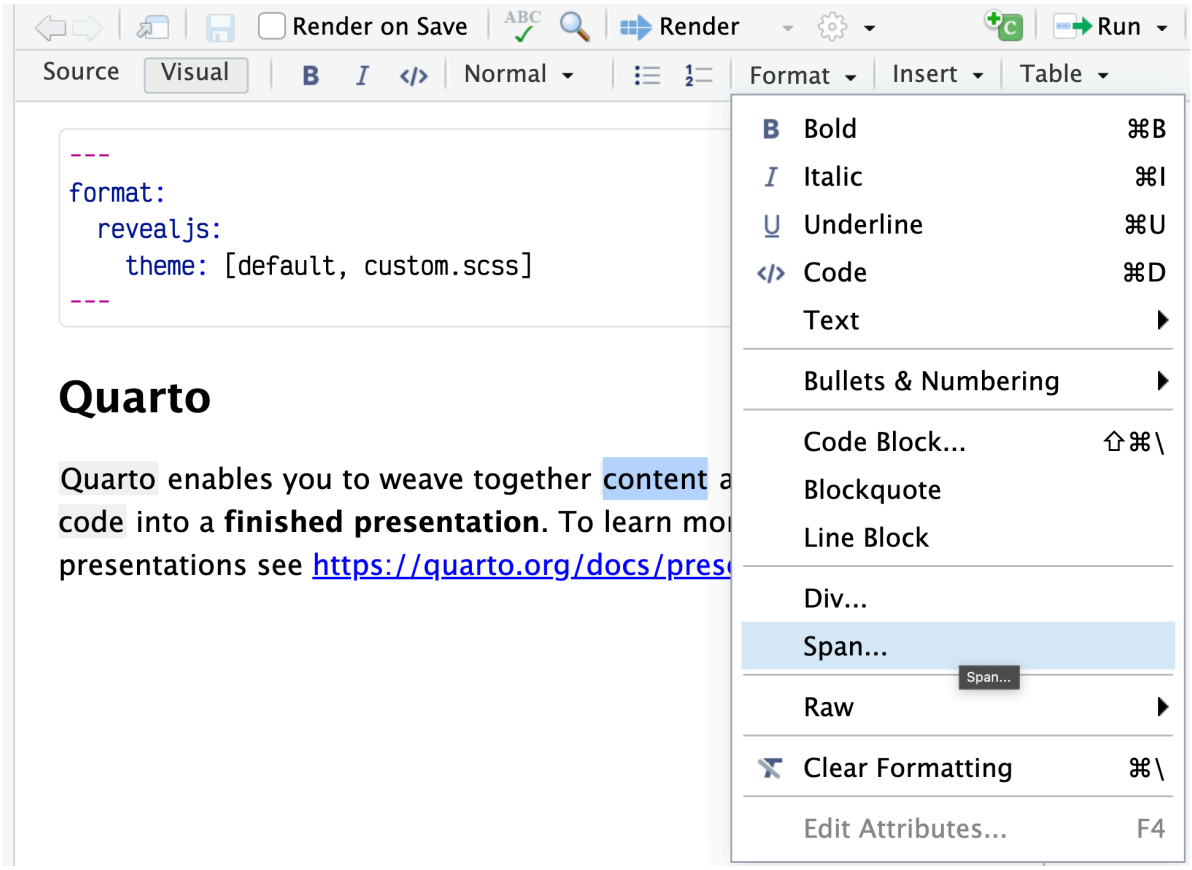
.blue {
  color: $theme-blue;
  font-weight: bold;
}

.yellow {
  color: $theme-yellow;
  font-weight: bold;
}
```

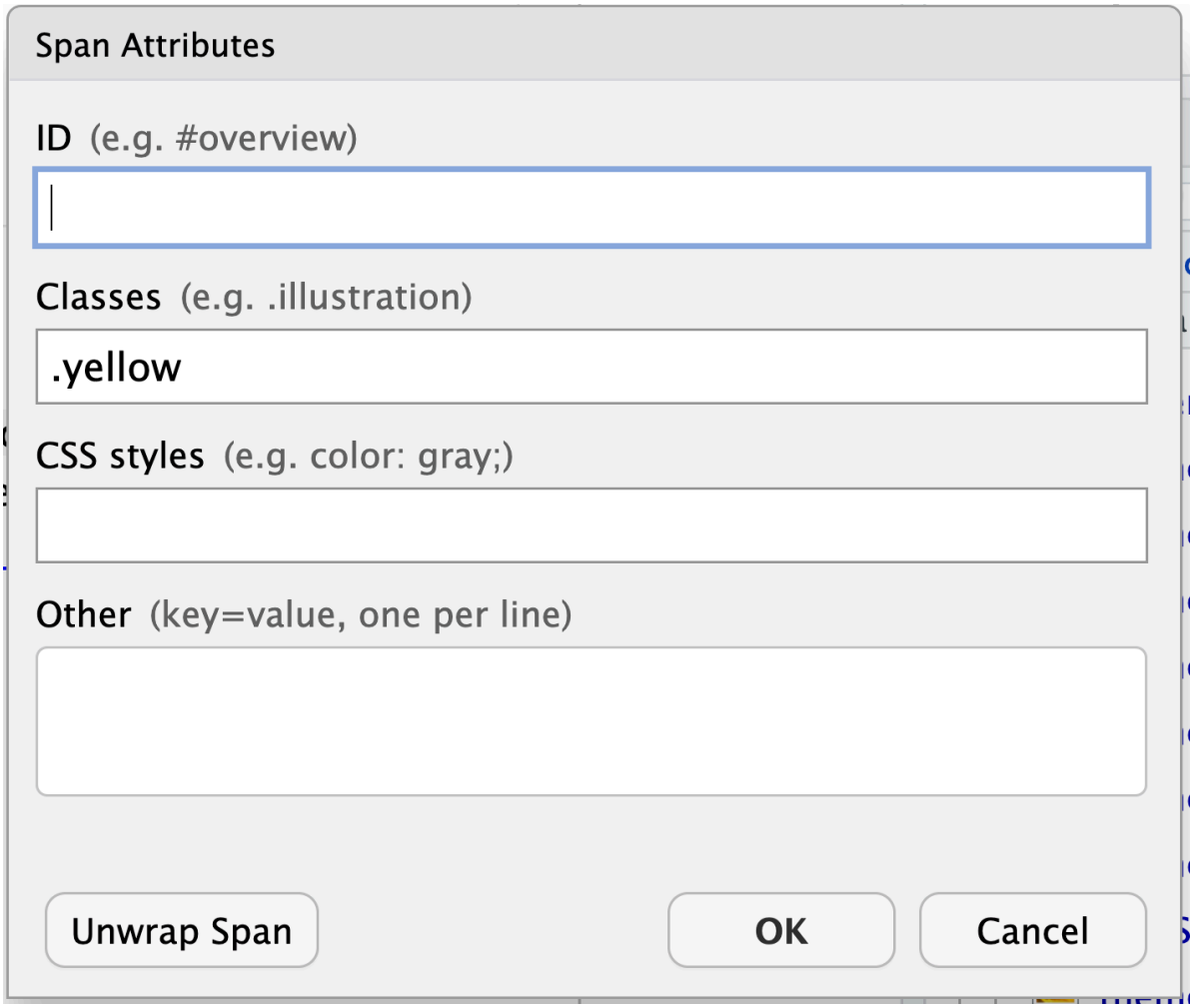
Now that we have our classes defined, you can either apply them using the visual editor in RStudio. Start by highlighting the text you want to apply the class to



Click "Format" and select "Span…" This will only apply the changes to the highlighted words, instead of the whole paragraph.



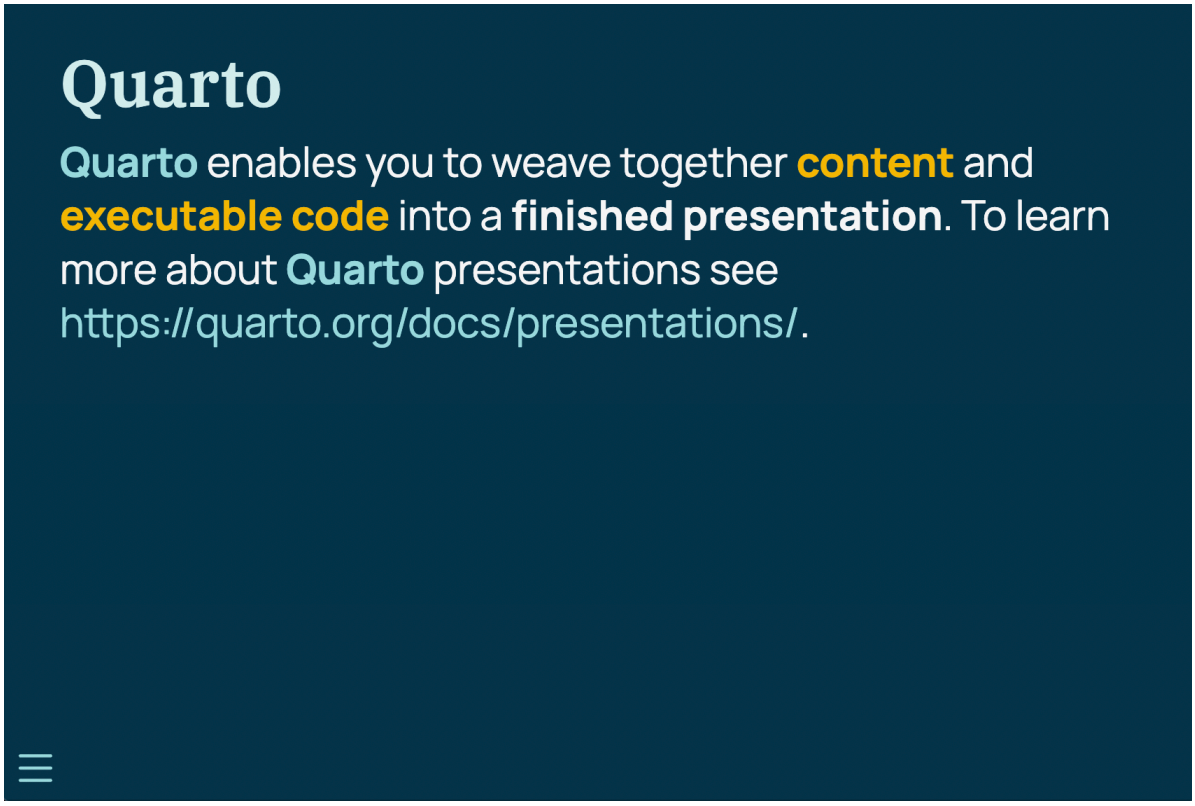
And then write in the class name in the "Classes" field. We see the word yellow by using the class ".yellow".



We can also apply these changes in the source editor by using the `[text]{.class}` syntax. I added (slightly excessive) highlighting to a couple of words. See below

```
[Quarto]{.blue} enables you to weave together [content]{.yellow} and [executable code]{.yello
```

And we get the following slide



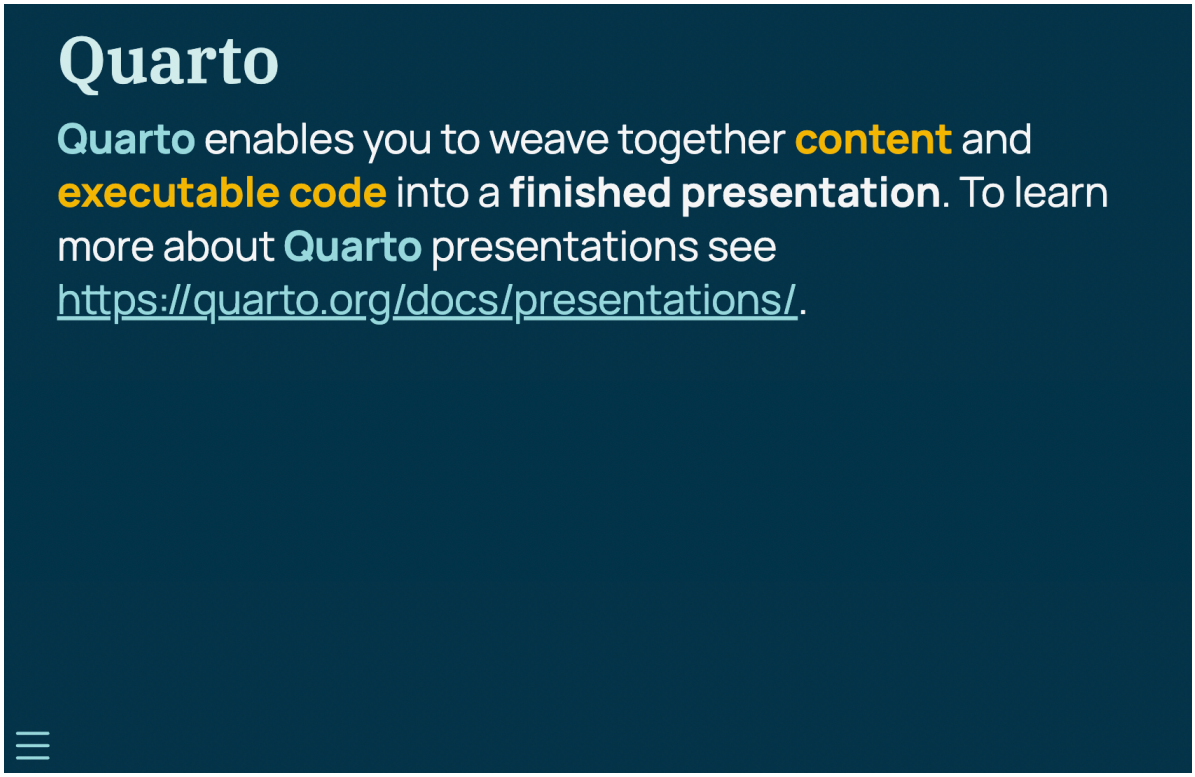
CSS classes were a game changer for my slide-making. It is a little bit more manual, but if you can write the CSS you can apply it to your slides which IMO is a super powerful tool.

There are also changes where you just want to modify the CSS directly, these changes should also be applied in the `scss:rules` section. For example, in our example so far, we have used the blue color both to color the links, and as a highlighting color. This is very confusing, so let us make sure that all links are underlined. We can make this happen by adding.

Note that CSS rules that target Reveal content generally need to use the `.reveal .slide` prefix to successfully override the theme's default styles.

```
.reveal .slide a {
  text-decoration: underline;
}
```

And the changes have been applied.



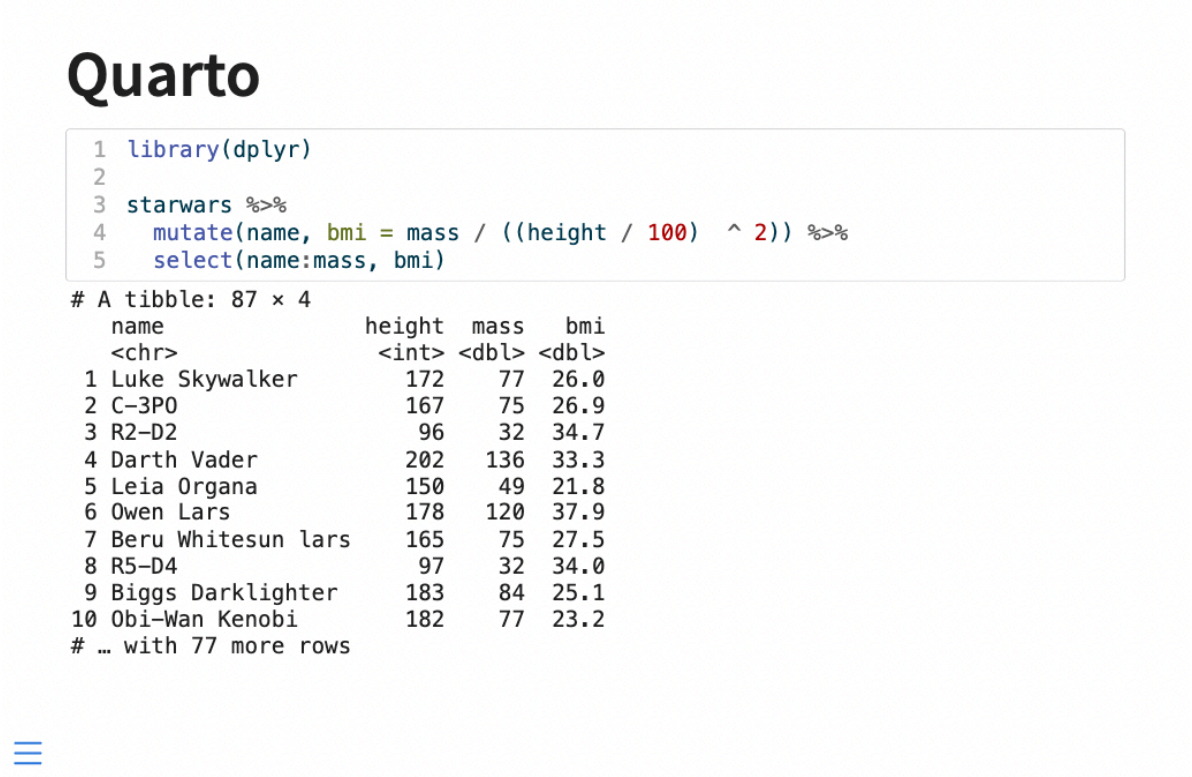
6.2 Style Code

to start with we need some code that produces some output, I'll use the following dplyr example, but any code and language will work the same. Notice how we are setting `echo: true` to include the source code in the output.

````
```r
#| echo: true
library(dplyr)

starwars %>%
  mutate(name, bmi = mass / ((height / 100)  ^ 2)) %>%
  select(name:mass, bmi)
```
````

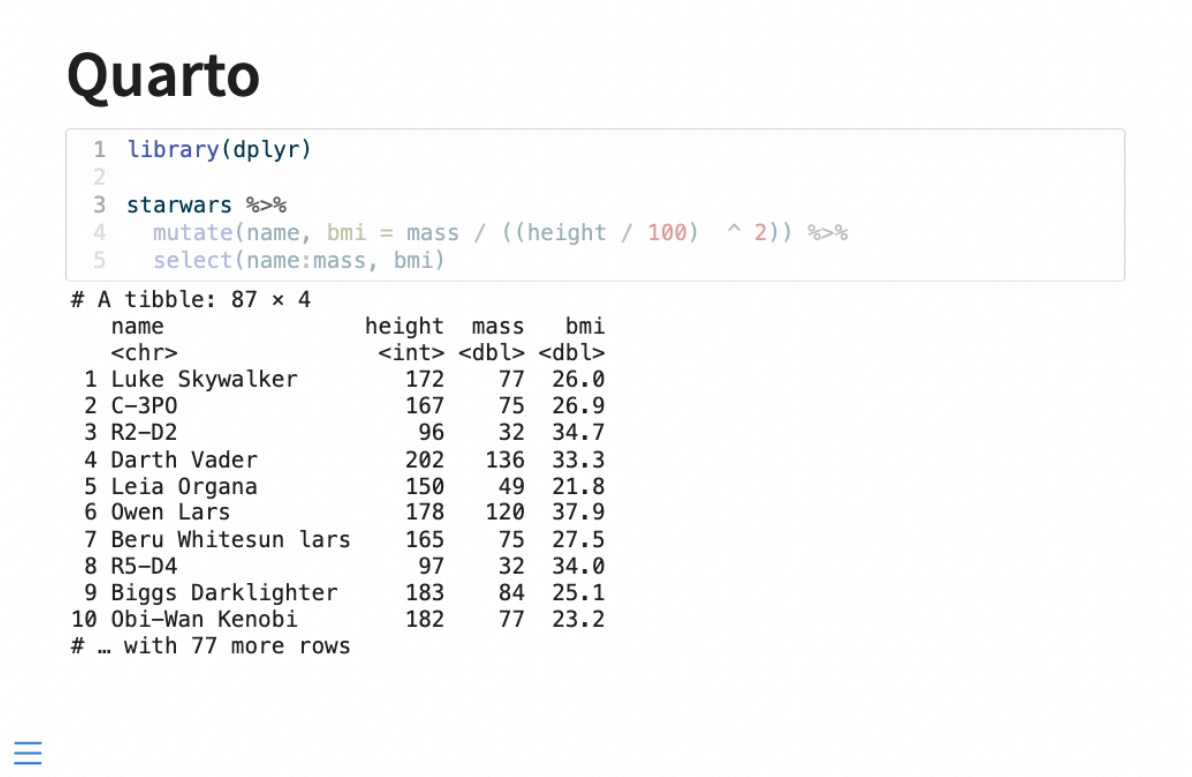
without any styling, this gives us the following slide



Now we have a couple of my favorite quarto arguments and then some CSS.

First, we have `code-copy` and `code-link` which both can be found in quarto revealjs reference. `code-copy` adds a little icon that when clicked copies the entire code block (super useful for teaching/reference slides) and `code-link` allows you to have downlit add clickable links to function in the code.

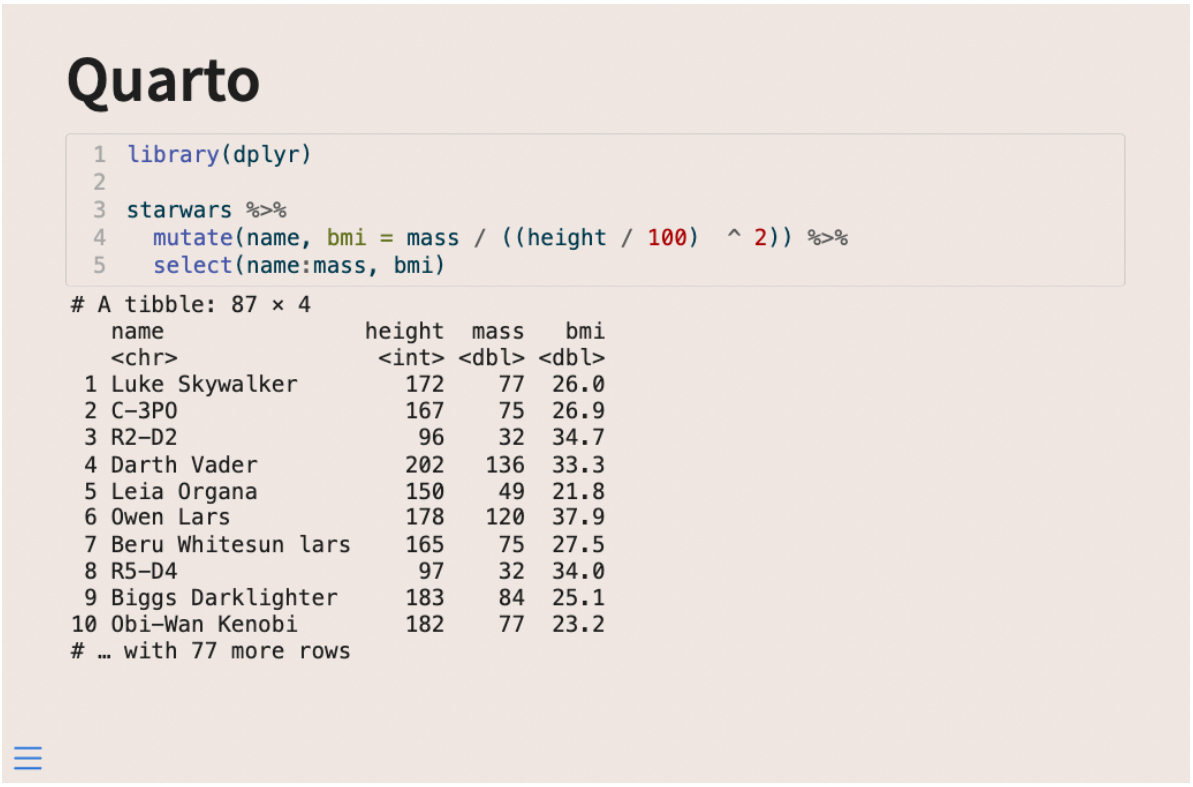
Lastly, I will point out `code-line-numbers`. This argument can either turn off or on the line numbers in the code block. This is nice and has a lot of benefits. Firstly you can turn off the numbering if you don't find that it is a good use of slide real estate. Secondly, you can specify line numbers to be put into focus. So setting `code-line-numbers: "1,3"` in the above example yields us the following output



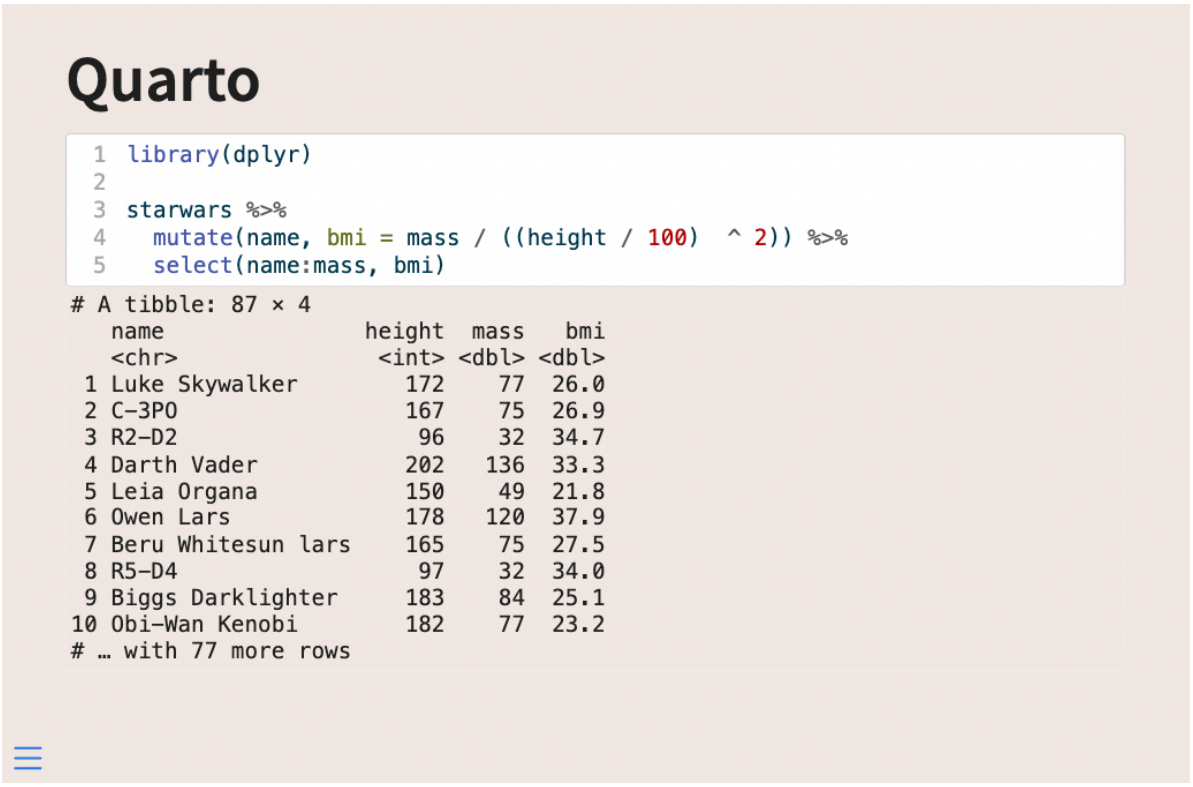
Which is a quick and easy way for you to direct the viewer's attention.

Next, let us talk about some styling. Before we do anything custom ourselves I should note that the 10 different built-in themes also affect the styling of the code and output, and these might give you what you need.

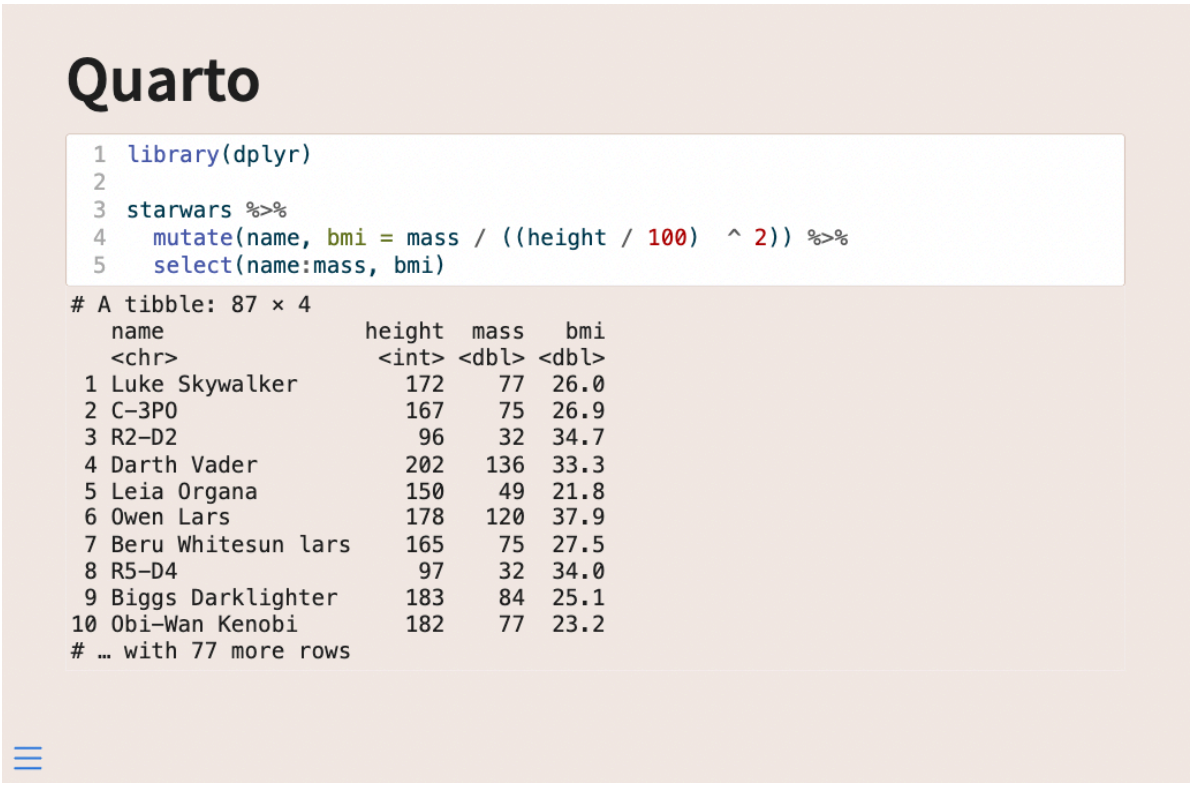
If you want to do a little bit more you need to get your hands dirty with some CSS and SCSS. Much like we did in the first blog post we are setting up a theme and applying it. I'm going with this new sandy background color I found.



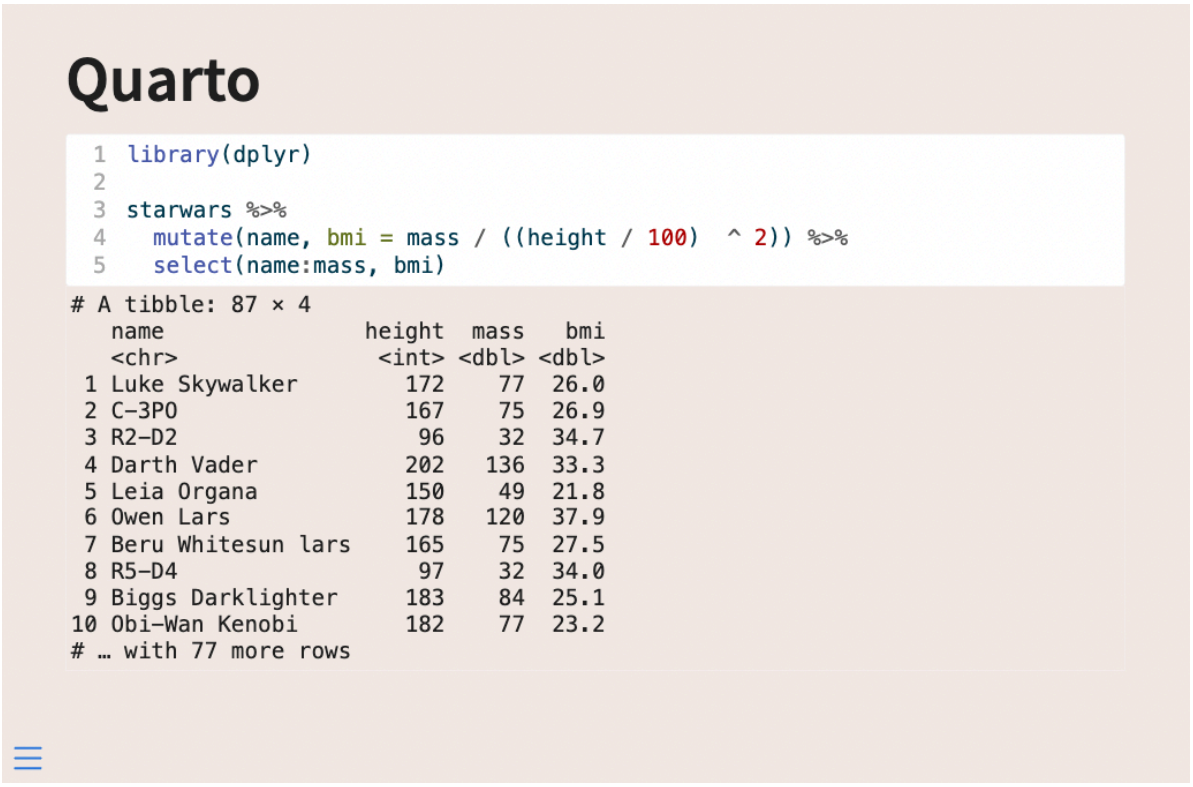
what I don't like about this slide right now is that the code block doesn't stand out that much. We have 3 major sass variables I want to introduce today; `$code-block-bg`, `$code-block-border-color`, and `$code-block-font-size`. `$code-block-bg` as the name suggests modifies the background of the code block, setting this color to white makes the code pop a little bit more



We can also change the border color with `$code-block-border-color`. This quantity defaults to being a lightened version of the main text color. We can instead set it to be a slightly darker version of the background color by setting `$code-block-border-color: darken($body-bg, 20%);` and getting this

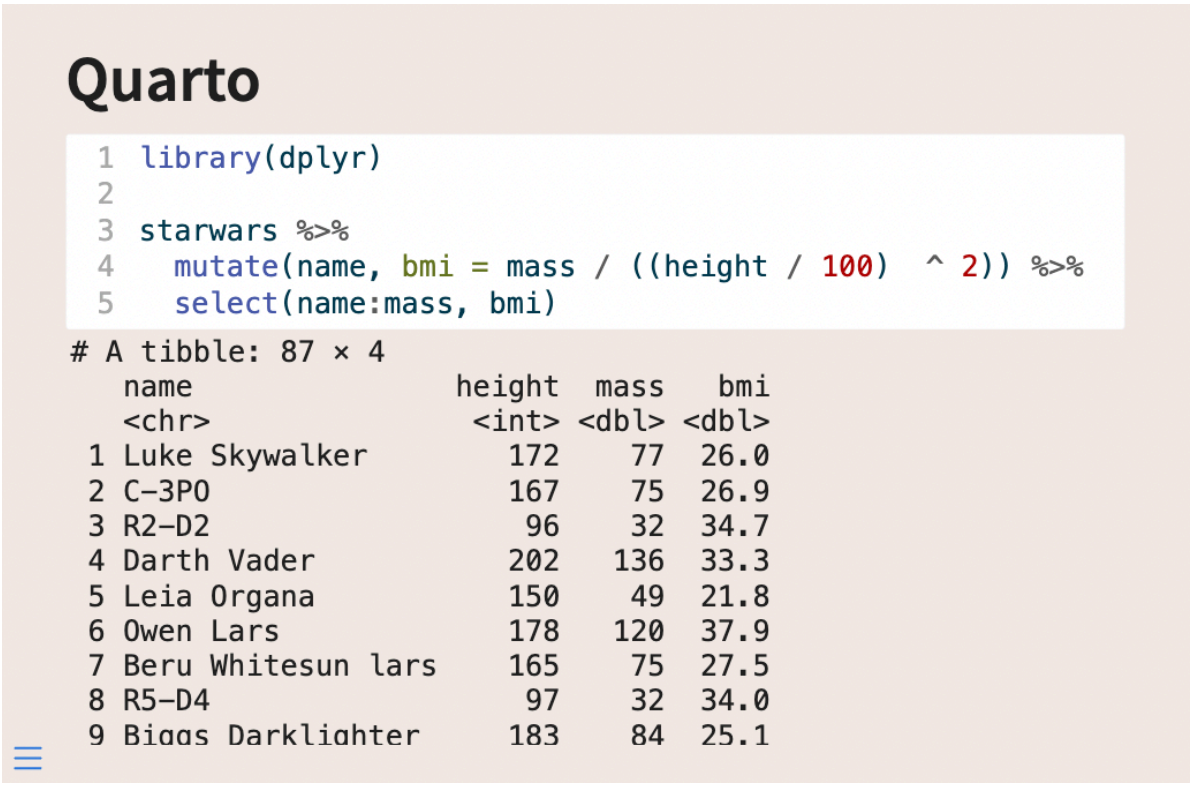


Or remove it entirely by setting it to be transparent with `$code-block-border-color: #00000000;`, for this look:



Lastly, we can change the font size with `$code-block-font-size`. While the size is pretty good right now, it is important that we know how to change it as it will depend on the fonts we use throughout the theme.

Below is an example where I turned up the size every so slightly



6.3 Style Output

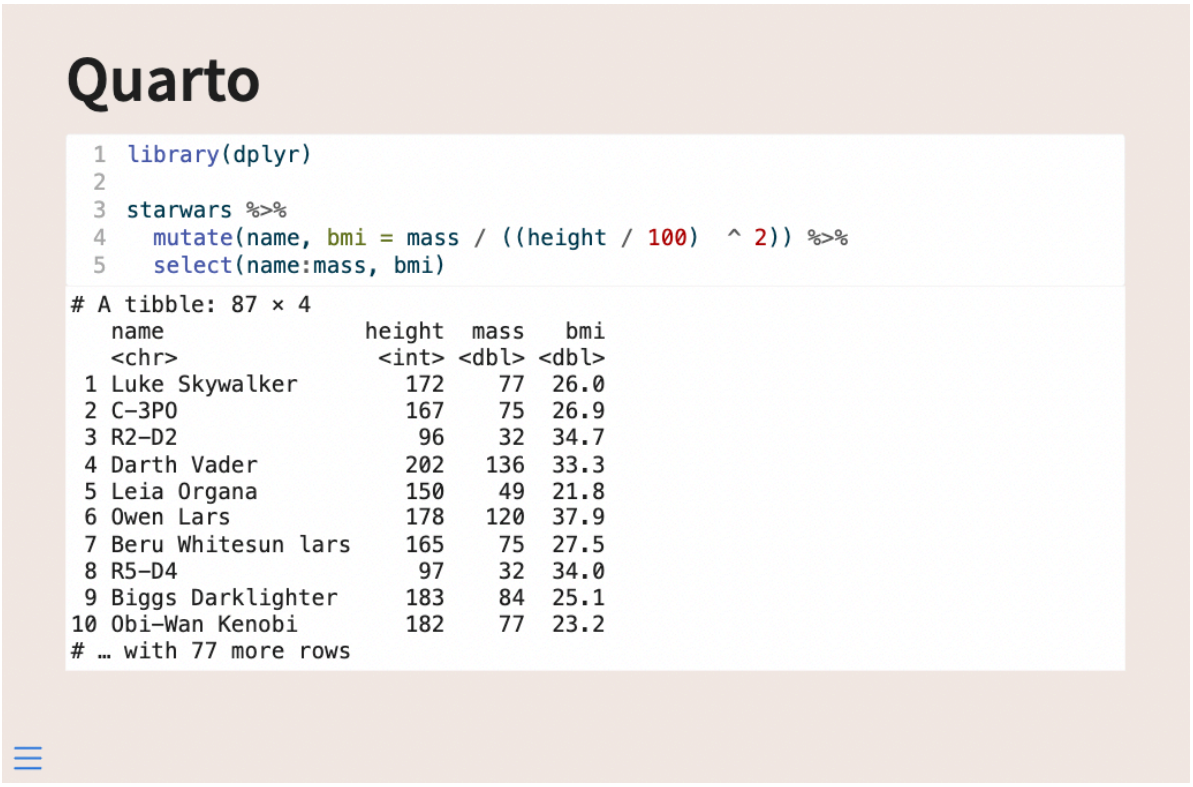
The next thing we want to look at is how to style the output. I found that adding styles to `.reveal pre code` did the trick in only changing the output, not the source.

Using the following code in our `.scss` file

```
/*-- scss:rules --*/

.reveal pre code {
  background-color: #FFFFFF;
}
```

Will give us the following output



Which is already Quite a bit better. For this, I would do either one of two things.

- Make the source and output appear to be one large box
- Separate the source and output more clearly

Let us see how we can do both of these things.

To make the source and output appear as one, we need to modify the source box, since it has a little bit of a border that is making them unequal. Setting `.reveal div.sourceCode {border: none}` should remove the existing border, and we can remove the slight rounded border as well by also adding `border-radius: 0px;`. This gives a full `.scss` as follows

```
/*-- scss:defaults --*/
$theme-sand: #F1EAE5;
$theme-white: #FFFFFF;

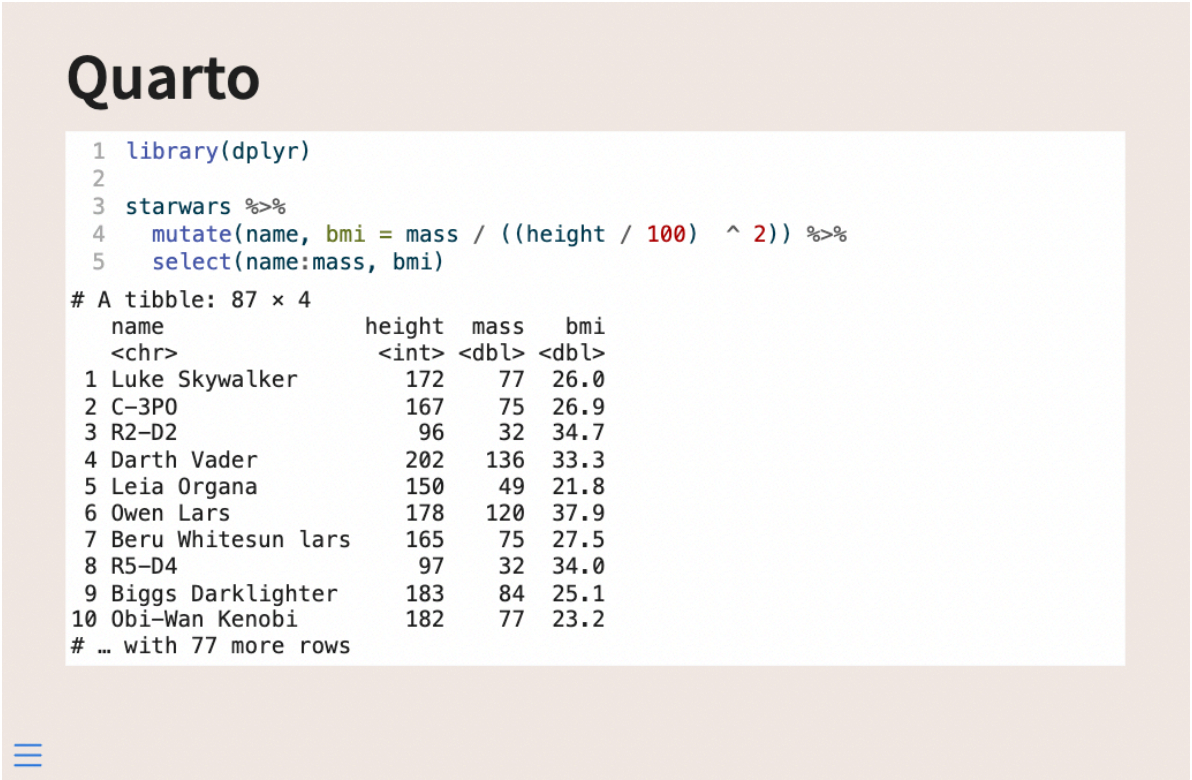
$body-bg: $theme-sand;
$code-block-bg: $theme-white;

/*-- scss:rules --*/
.reveal pre code {
```

```
  background-color: $theme-white;
}

.reveal div.sourceCode {
  border: none;
  border-radius: 0px;
}
```

This gives us the following slides



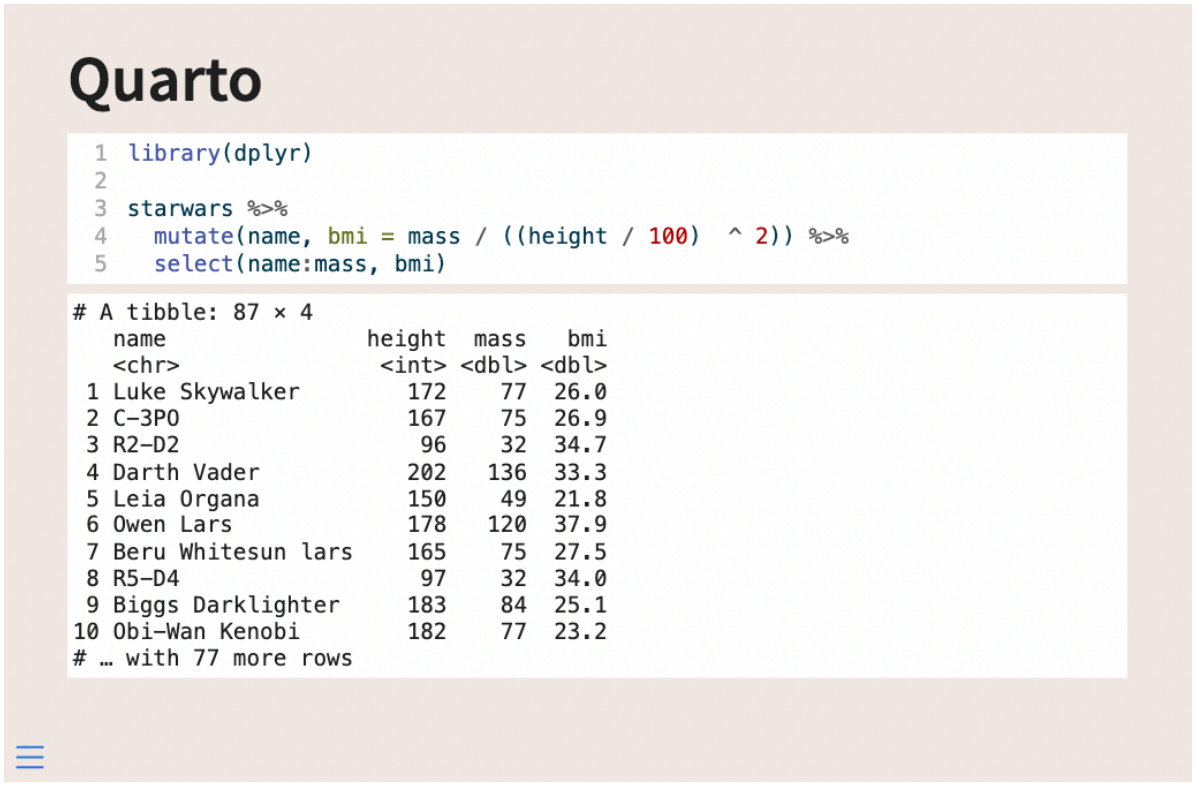
If we want some space, we can add `margin-bottom: 10px !important;` to the `.reveal div.sourceCode` specification, making it so there will be 10 pixels worth of space below the source, which gives some spacing between the two boxes. You need the `!important` to overwrite a more general style specification.

Making it so that part of the `.scss` file now looks like this

```
.reveal div.sourceCode {
  border: none;
  border-radius: 0px;
```

```
  margin-bottom: 10px !important;
}
```

And you now have a little bit of air between the boxes



6.4 Change highlighting theme

One thing we haven't looked at yet is how to change the highlighting theme. We will see how we can do that here. This is an often overlooked part of a presentation, that if you spend some time can tie your presentation together.

This is being controlled in the `highlight-style` argument. I would generally recommend that you set this option globally in your document by including it in your YAML file like so

```
---
format:
  revealjs:
    theme: [default, custom.scss]
```

```
highlight-style: "dracula"
---
```

Which would result in the following output

> **Warning**
>
> Some of the themes such as the `"dracula"` theme come with a built-in background color. If you set `$code-block-bg` in your `.scss` file it will be overwritten.

```
{
    "text-color": null,
    "background-color": "#f8f8f8",
    "line-number-color": "#aaaaaa",
    "line-number-background-color": null,
    "text-styles": {
        "Other": {
            "text-color": "#8f5902",
            "background-color": null,
```

```
            "bold": false,
            "italic": false,
            "underline": false
        },
        "Attribute": {
            "text-color": "#c4a000",
            "background-color": null,
            "bold": false,
            "italic": false,
            "underline": false
        },
```

There are many different highlighting themes that come shipped with Quarto. The ones with `-dark` and `-light` postfixes are themes that will adjust depending on whether your slides are dark or light.

If none of these are what you want, you can create your own. I would suggest taking one of the themes you like from the list linked above and modifying it to your liking. Copy that file into the same directory as your slides, and point to the file name instead. Like so:

```
---
format:
  revealjs:
    theme: [default, custom.scss]
highlight-style: "darkula.theme"
---
```

If your slides use both dark and light backgrounds you will likely need two sets of highlighting theme files, which you can set with:

```
---
format:
  revealjs:
    theme: [default, custom.scss]
highlight-style:
  light: light.theme
  dark: dark.theme
---
```

I slowly modified the base theme, using the Working with Color Themes documentation to help me figure out what the different names mean.

Another thing that can aid, is opening the developer tools in your browser and hovering over the kind of elements you care about. Below I am doing just that, and I see that the pipe is of class `sc`

From that, you can look at this list and see that it is a `SpecialChar` and you can modify your theme accordingly.

- ot: **Other**
- at: **Attribute**
- ss: **SpecialString**
- an: **Annotation**
- fu: **Function**
- st: **String**
- cf: **ControlFlow**
- op: **Operator**
- er: **Error**
- bn: **BaseN**
- al: **Alert**
- va: **Variable**
- pp: **Preprocessor**
- in: **Information**
- vs: **VerbatimString**

- wa: **Warning**
- do: **Documentation**
- ch: **Char**
- dt: **DataType**
- fl: **Float**
- co: **Comment**
- cv: **CommentVar**
- cn: **Constant**
- sc: **SpecialChar**
- dv: **DecVal**
- kw: **Keyword**

With these changes in place, I get to the following slides

For this theme, I also updated `.reveal pre code` to change the background color and the text color of the output to match the rest of the theme.

You generally don't need to update all the items in the theme as you are unlikely to use all of them. I tend to update as I go, only updating the classes that I use.

6.5 Use fonts with ligatures

I just added some new code to a slide

and this code uses some multi-character symbols like `!=` and `|>`. These and many like them are common all over the different programming languages, so much so that people have created special fonts with ligatures to make them prettier.

One such font is FiraCode.

And we can add these types of fonts to our slides as well! Start by downloading the font from the site, and copy over a `.woff` and `.woff2` file to your slide directory. I selected `FiraCode-Regular.woff` and `FiraCode-Regular.woff2`.

In the `/*-- scss:defaults --*/` part of our `.scss` file, we are going to add the following code. This is done to register the font family from the files we have included and to have it selected for use for monospace fonts.

```
@font-face {
    font-family: 'FiraCode';
    src: url('../../../../../FiraCode-Regular.woff2') format('woff2'),
         url('../../../../../FiraCode-Regular.woff') format('woff');
}

$font-family-monospace: 'FiraCode';
```

You might have noticed some ugliness with the `../../../../../`s. To my knowledge, this is the best way of using a local font-face from a file.

With all of that, we now have beautiful ligatures

6.6 Step up your style

Every time I have shared some code with carbon.now.sh people go crazy because it always looks so good. And I agree, the defaults are really really nice.

So we are going to replicate this. This is going to be a little more involved, but if you don't have that much code on your slides it will be a nice touch!

What I do in this case is use reprex to generate the code and output as one, I paste that into my chunk and turn off evaluation with `#| eval: false`. I do this because then I don't have to worry about styling the output to match.

I switch over to using the `"nord"` highlight palette. It has this weird mistake because it thinks that the opening bracket is an error.

```
diamonds ▷
  count(color, cut, sort = TRUE)
# A tibble: 35 × 3
   color cut              n
   <ord> <ord>      <int>
 1 G     Ideal       4884
 2 E     Ideal       3903
 3 F     Ideal       3826
```

I could copy over the `.theme` file and modify it in the right place, but I'm lazy and modify it directly in the `.scss` file instead by adding which will fix the color and remove the underline

```
code span.er {
  color: #ebcb8b;
  text-decoration: none;
}
```

The line numbers are not too important this code chunk so I set `#| code-line-numbers: false`. Then I start working on styling the source div.

```
.reveal div.sourceCode {
  border: none;
  border-radius: 5px;
  margin-bottom: 10px !important;
  box-shadow: 0 20px 47px rgb(0 0 0 / 55%);
  width: fit-content;
  margin: auto !important;
}
```

The main thing that changes is that I brought back the `border-radius`. I added some `box-shadows`. These things alone make a huge difference in appearance.

Next, I changed `width: fit-content;`, this makes it so the width of the div changes with the width of the code, the code I have right now isn't that wide and it looked a little lopsided. Lastly, I set `margin: auto !important;`, which centers the div so it doesn't cling to the left side.

The very final thing I changed is adding some padding to the code, this is the space between the code itself and the inside border of the div, I did this by adding

```
.reveal div.sourceCode pre code {
  padding: 25px;
}
```

All of this results in the following slide:

6.7 Style menu button

The menu button you see in the lower left-hand side of the slide. Styling it can be done by setting the `$link-color` sass variable. If you want a different icon, or have it colored differently than `$link-color` you need to specify it directly as the color is hardcoded into the svg. The icon is specified as the background image of `.reveal .slide-menu-button .fa-bars::before`.

```
.reveal .slide-menu-button .fa-bars::before {
background-image: url('data:image/svg+xml,<svg xmlns="http://www.w3.org/2000/svg" width="16"
}
```

Tthe color is specified by the `fill="rgb(42, 118, 221)"` part of the svg. But since this is an image, we can use whatever image we want.

```
.reveal .slide-menu-button .fa-bars::before {
background-image: url('https://cdn-icons-png.flaticon.com/512/2163/2163350.png') !important;
}
```

qmd scss

7 SCSS

7.1 Using `Maps` to store theme colors

Before I would have specified a color palette theme as

```
$theme-red: #FA5F5C;
$theme-blue: #394D85;
$theme-darkblue: #13234B;
$theme-yellow: #FFF7C7;
$theme-white: #FEFEFE;
```

But using a map it turns into the following

```
$colors: (
  "red": #FA5F5C,
  "blue": #394D85,
  "darkblue": #13234B,
  "yellow": #FFF7C7,
  "white": #FEFEFE
);
```

> **Note**
>
> There is a difference between `(key: value)` and `("key": value)` in SASS. For consistency, I also use quoted strings as the keys in a map.

Instead of having multiple values representing our colors we now just have one `$colors`. By themselves, maps aren't valid CSS and don't do anything once the SASS compiles. The next sections will show how we can use these maps more efficiently than my previous approach.

7.2 Using `Functions` to pull out theme colors

We where using these colors to change a lot of things, including the major Sass Variables revealjs sass variables. Before we used a map, it would look something like this:

```
$body-bg: $theme-yellow;
$link-color: $theme-blue;
$code-color: $theme-blue;
$body-color: $theme-darkblue;
```

but we can't do that directly with a map. There are built-in functions to extract values from a map, namely the function map-get(). Using that we can rewrite the above as

```
$body-bg: map-get($colors, "yellow");
$link-color: map-get($colors, "blue");
$code-color: map-get($colors, "blue");
$body-color: map-get($colors, "darkblue");
```

And while that is all good, I find it a little nicer to have a helper function to do this.

Functions in SASS are written as below. You can use as many arguments as you want.

```
@function name($arg1, $arg2) {
  @return $arg1 + $arg2;
}
```

The helper function I wrote is a light wrapper around `map-get()` to avoid having to write `$colors`.

```
@function theme-color($color) {
  @return map-get($colors, $color);
}
```

And we now have the final rewrite.

```
$body-bg: theme-color("yellow");
$link-color: theme-color("blue");
$code-color: theme-color("blue");
$body-color: theme-color("darkblue");
```

> **Note**
>
> Note that we are using `theme-color("yellow")` instead of `theme-color(yellow)` because we used quoted strings in the map. Using unquoted strings all around gave me false positives in my IDE as it interpreted `yellow` inside `theme-color()` as `#FFFF00` instead

> of my theme value.

7.3 Using `@each` to automatically create classes

To add a splash of color, or as used in highlighting, I would create a lot of CSS classes like so:

```
.text-red {
  color: $theme-red;
}
.text-yellow {
  color: $theme-yellow;
}
.text-blue {
  color: $theme-blue;
}

.bg-red {
  background-color: $theme-red;
}
.bg-yellow {
  background-color: $theme-yellow;
}
.bg-blue {
  background-color: $theme-blue;
}
```

Which is all fine and dandy until you also want a class for underlining. It becomes a lot of copy-pasting and changing a couple of names. And that is not to mention the trouble you run into when you decide to add a new color into the mix halfway through your slides.

This is when I discovered interpolation and the moment I realized maps were worth it. Interpolation is done by using `#{}` in some code, meaning that if `$favorite-color: "blue"` then `.text-#{$favorite-color} {}` turns into `.text-blue {}`. SASS provides the action `@each` to loop over all the key and value pairs of our map. So we can rewrite the creation of the above classes as this:

```
@each $name, $color in $colors {
  .text-#{$name} {
    color: $color;
```

```
  }

  .bg-#{$name} {
    background-color: $color;
  }
}
```

And this is the beauty of maps. If I want to add a new color to my slides, I just have to add it to the `$colors` map. If I want to add a new set of classes, I just have to write it once inside the `@each` statement.

7.4 Using `@mixin` to avoid repeating code

With the following code

```
@mixin background-full {
  background-size: cover;
  background-position: center;
  background-repeat: no-repeat;
}

.theme-slide1 {
  &:is(.slide-background) {
    background-image: url('../../../../../assets/slide1.svg');
    @include background-full;
  }
}
.theme-slide2 {
  &:is(.slide-background) {
    background-image: url('../../../../../assets/slide2.svg');
    @include background-full;
  }
}
.theme-slide3 {
  &:is(.slide-background) {
    background-image: url('../../../../../assets/slide3.svg');
    @include background-full;
  }
}
```

Copy-pasting this around when you have 10-15 images is such a pain. But as you properly have noticed, interpolation would be a perfect solution to this problem, and you would be correct! We use a `@mixin` to create to create a way to create many classes:

```
@mixin theme-slide($number) {
  .theme-slide#{$number} {
    &:is(.slide-background) {
      background-image: url('../../../../../assets/slide#{$number}.svg');
      @include background-full;
    }
  }
}

@include theme-slide(1);
@include theme-slide(2);
@include theme-slide(3);
```

It becomes quite a bit tighter! But we can do better because SASS also has `@for` loops.

```
@mixin theme-slide($number) {
  .theme-slide#{$number} {
    &:is(.slide-background) {
      background-image: url('../../../../../assets/slide#{$number}.svg');
      @include background-full;
    }
  }
}

@for $i from 1 through 3 {
  @include theme-slide($i);
}
```

you can now add more background images with ease as long as you are careful when naming them.

7.5 Using nested loops to create classes

In our first example, I want a fun gradient shadow/highlight effect for text. We can get that effect using something like following

```
background-image: linear-gradient(90deg, yellow, blue);
background-size: 100% 42%;
background-repeat: no-repeat;
background-position: 0 85%;
width: fit-content;
```

I want it to work on both inline text and as a way to handle the header, the selectors for that would be `span.my-class` and `.my-class > h2` respectively.

```
span.my-class, .my-class > h2 {
  background-image: linear-gradient(90deg, yellow, blue);
  background-size: 100% 42%;
  background-repeat: no-repeat;
  background-position: 0 85%;
  width: fit-content;
}
```

And them we fill in the rest, using the `@each` command twice nestedly over a map of colors.

```
/*-- scss:defaults --*/

$colors: (
  "red": #FFADAD,
"orange": #FFD6A5,
  "yellow": #FDFFB6,
  "blue": #aad2e7,
  "purple":#b4addd
);

/*-- scss:rules --*/

@each $name1, $col1 in $colors {
  @each $name2, $col2 in $colors {
    span.hl-#{$name1}-#{$name2}, .hl-#{$name1}-#{$name2} > h2 {
      background-image: linear-gradient(90deg, $col1, $col2);
      background-size: 100% 42%;
      background-repeat: no-repeat;
      background-position: 0 85%;
      width: fit-content;
    }
  }
}
```

I know we are creating some non-interesting classes such as `.hl-yellow-yellow` but for what we are doing, the tradeoff between avoiding them and how little it impacts us to have them. I think it is a worthwhile tradeoff.

> **Important**
>
> The slides in this post are interactive, advance them to see the other classes.

qmd scss

> **Note**
>
> You don't need these compound classes for everything. For example, the class `.hl-green-bold` isn't going to be useful as you could just as easily create `.hl-green` and `.bold` separately. This trick works best when two elements are used together in a tightly coupled way, such as in gradients.

For our second example, we are continuing with the gradients, but instead trying to apply them to the background. My goal was to add a gradient line to the right side of the slide.

I was able to create that effect, by layering 2 gradients on top of each other. The first gradient contained the two colors I was interested in, and the names of the class. The second layer, which I placed on top, goes from white to transparent. I set up the transition between those two colors to be super sharp, resulting in the effect you see below

qmd scss

since we are doing something interesting, we could also have used a separate `$colors` map just for this effect to not interfere with what else we are doing.

Part II

Content

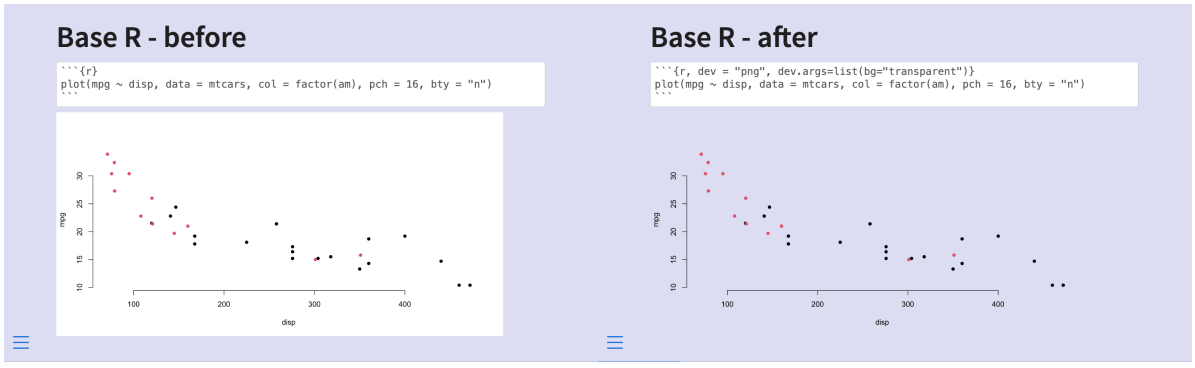
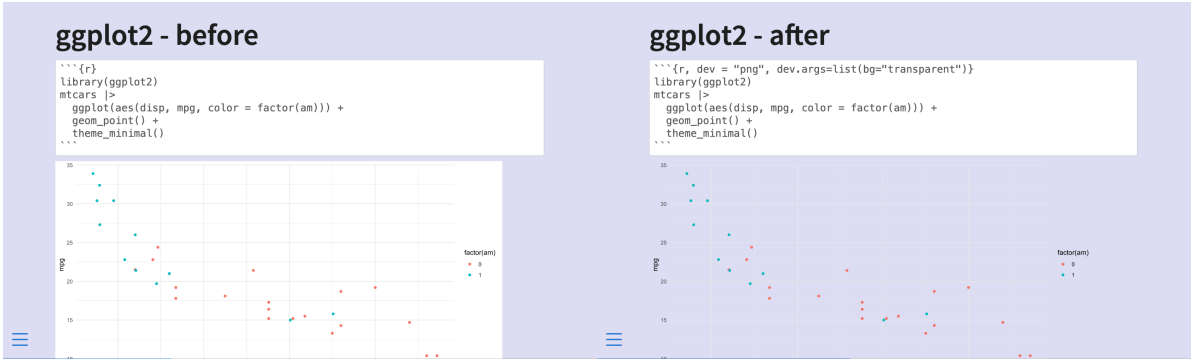
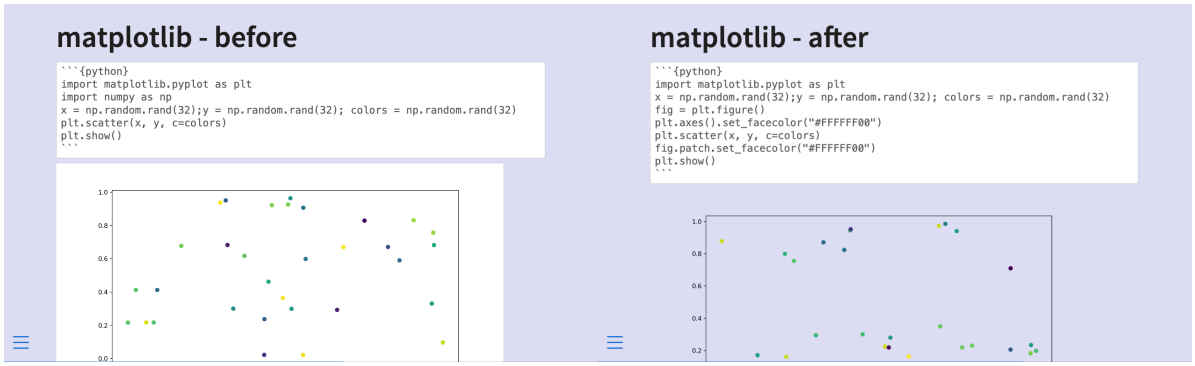
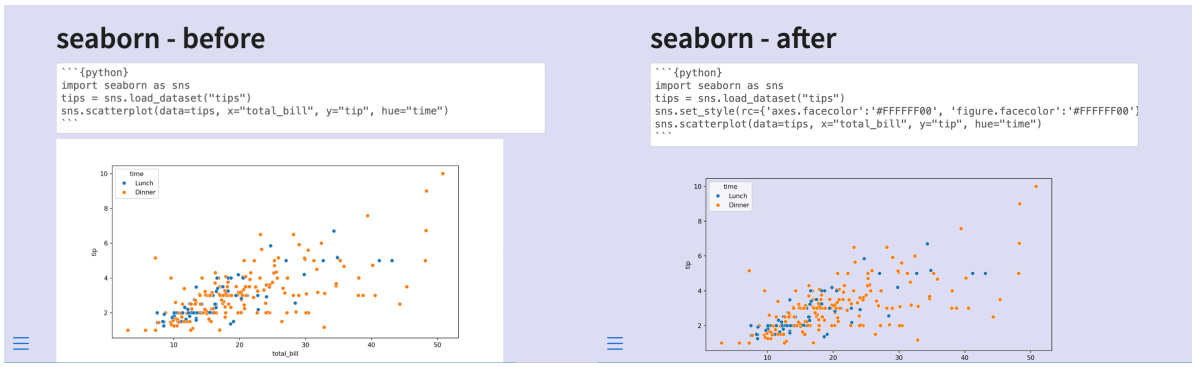
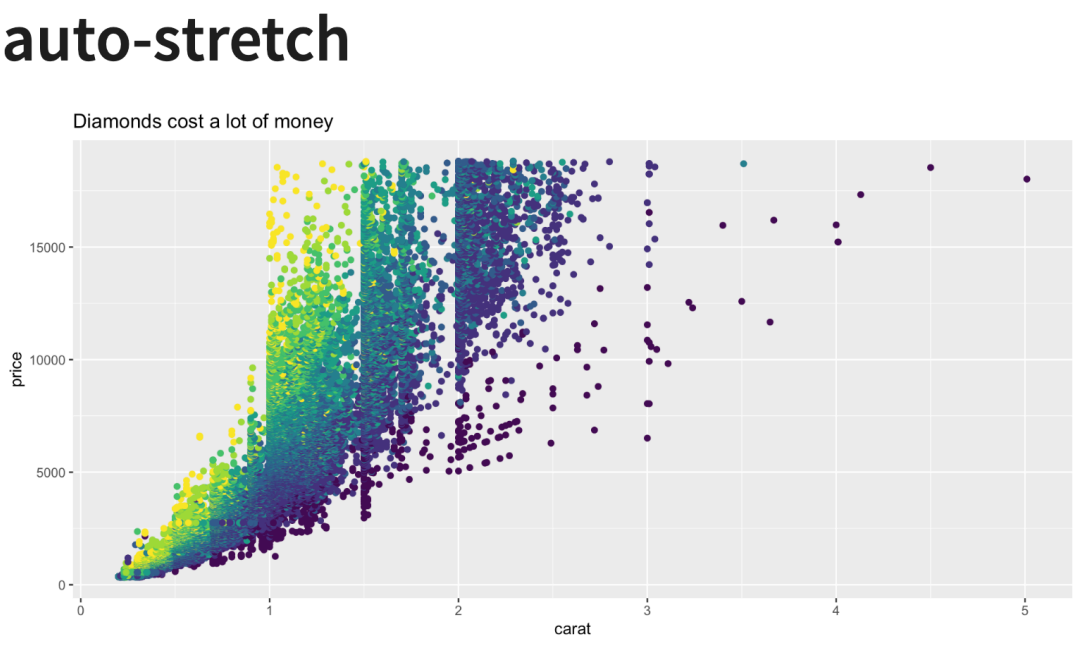
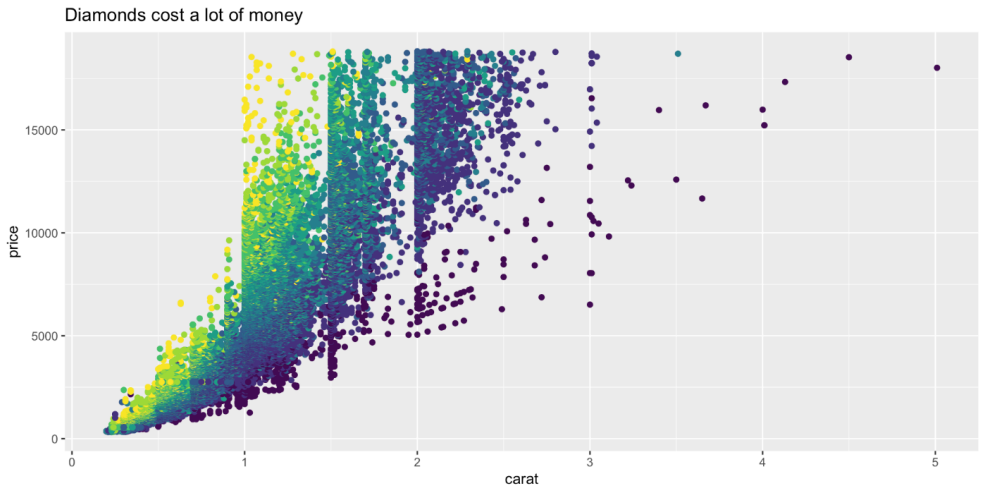
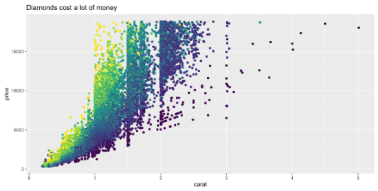
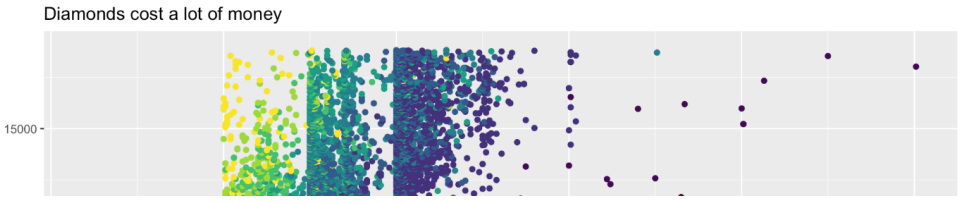
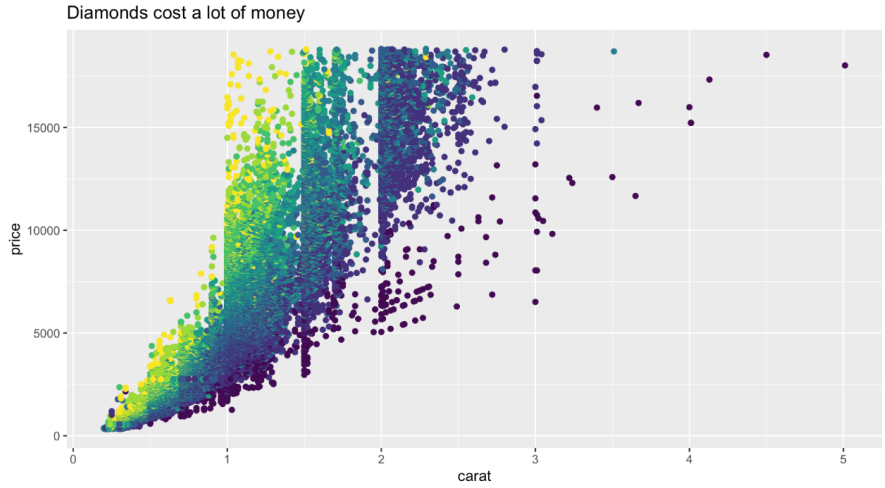
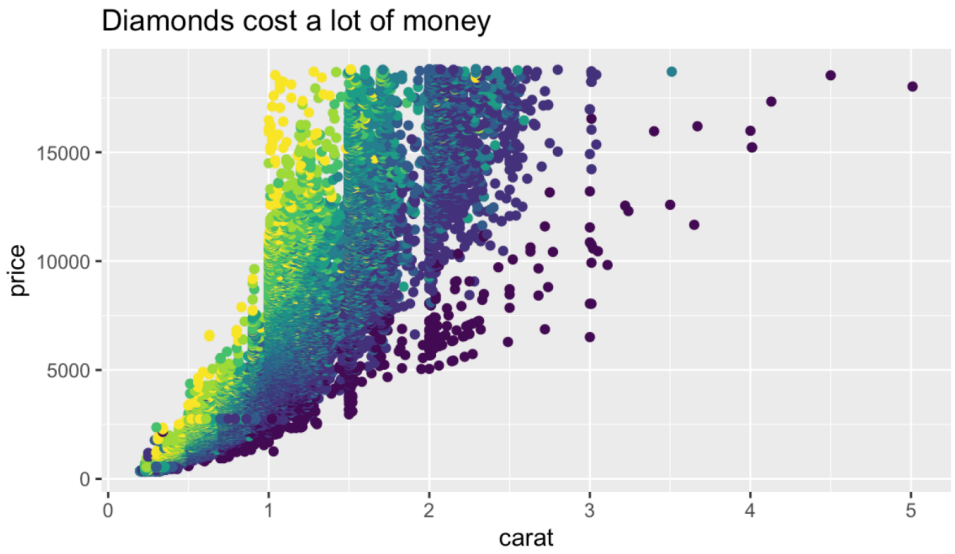
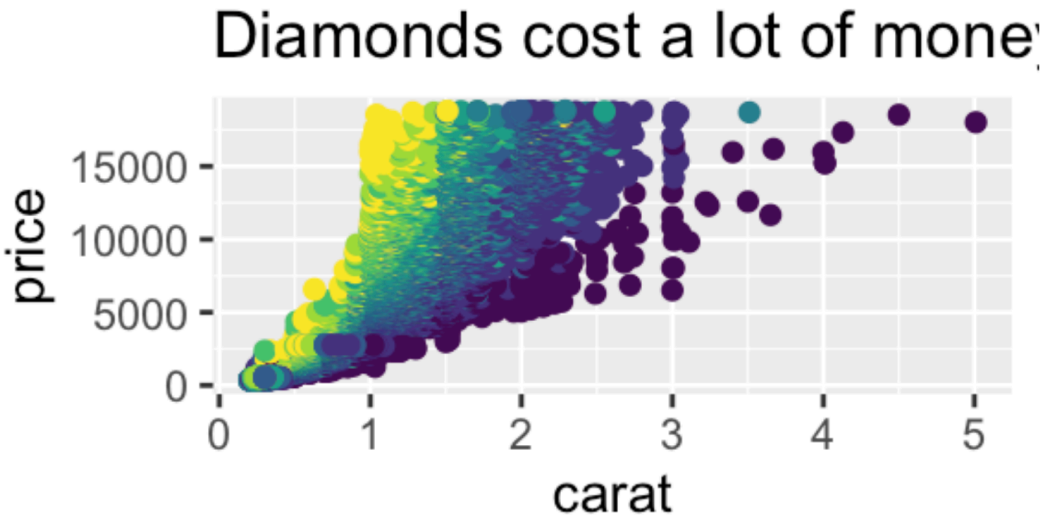
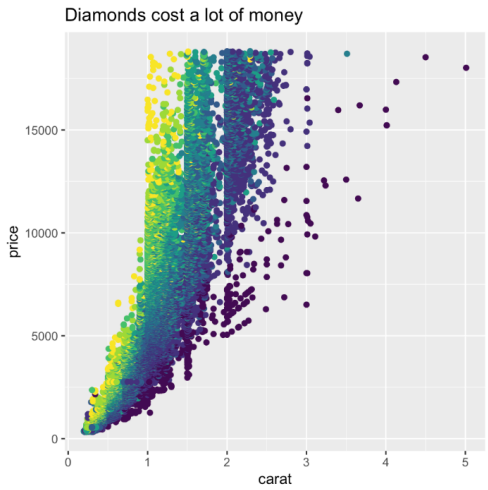
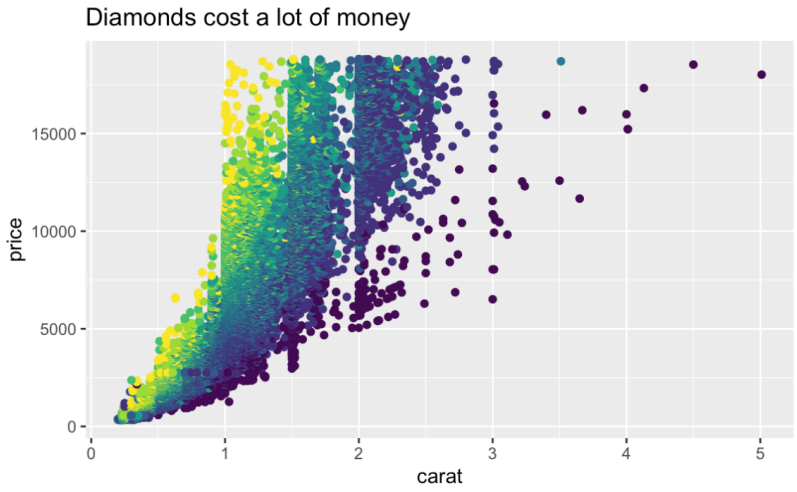
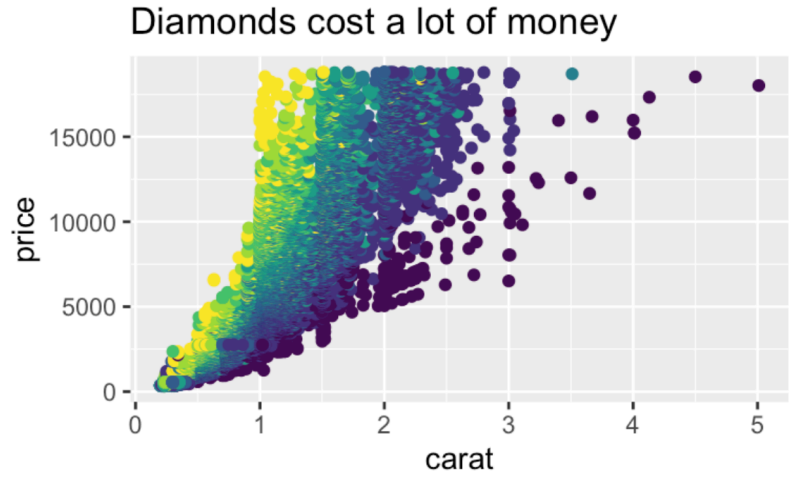
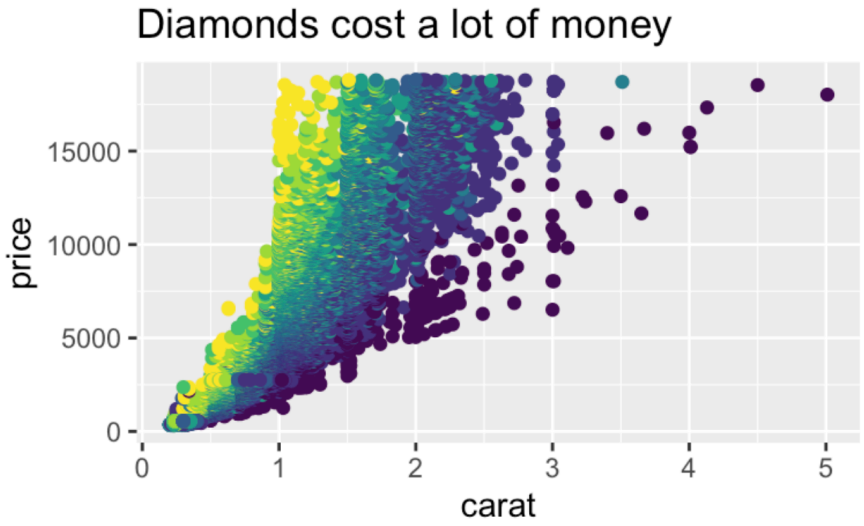
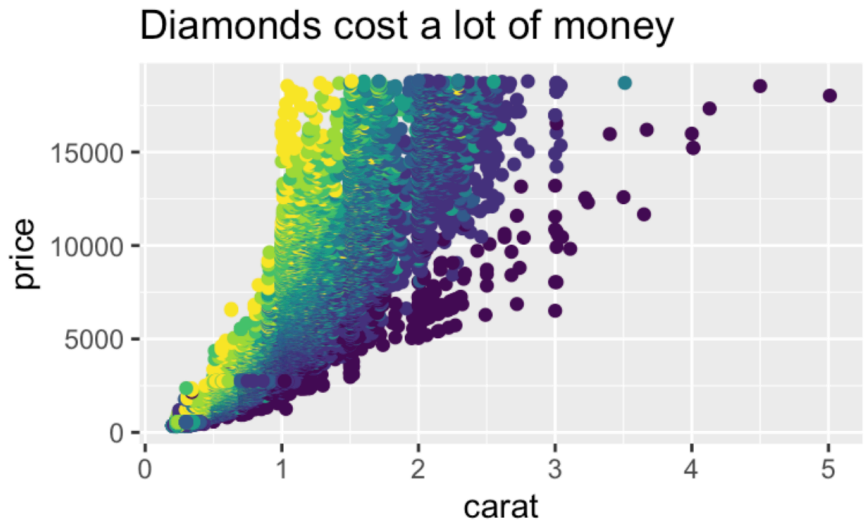
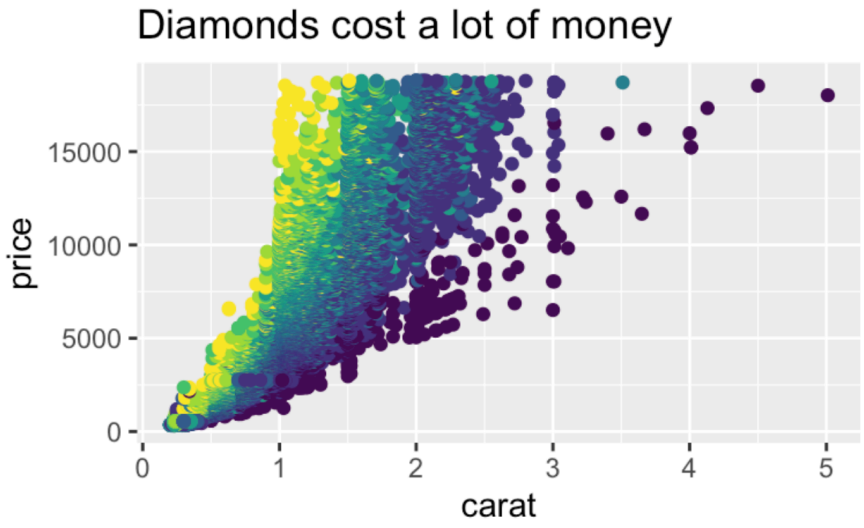
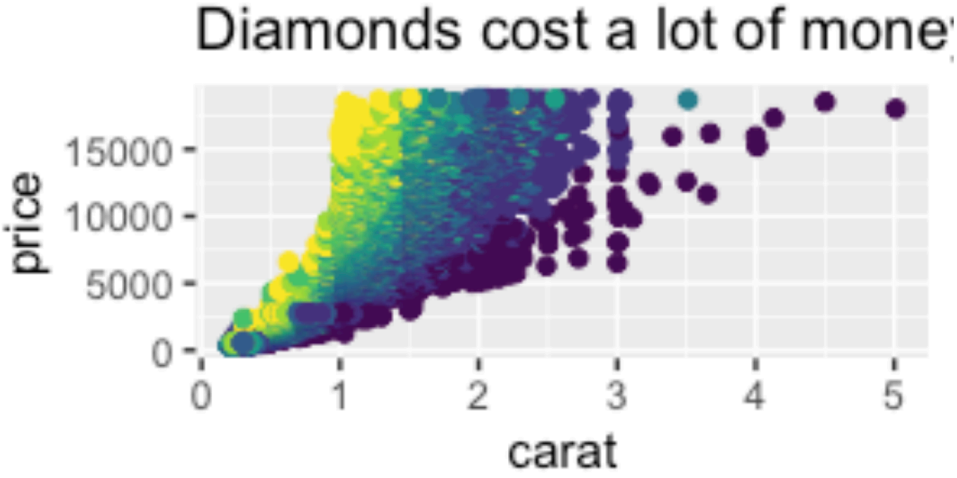
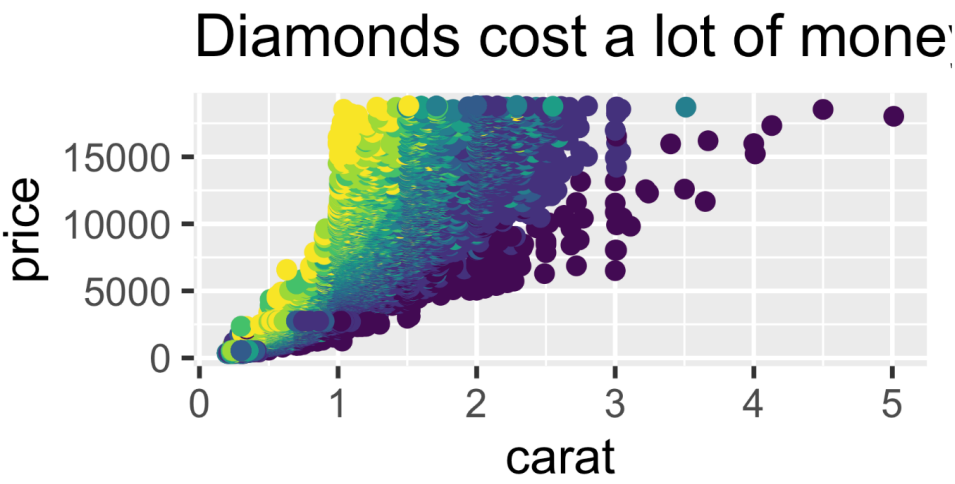
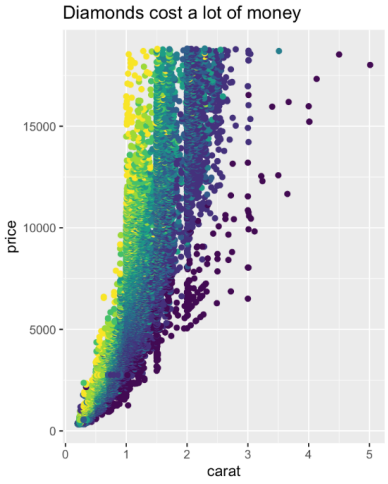
8 Elements

8.1 Showing quarto code

This one isn't as much a slidecrafting tip, as it is a quarto tip! If you are showing how to do something in Quarto using Quarto you need this tip. In essence what we are working with are unexcuted blocks.

Adding a `markdown` cell around what you want to show. Important to use more ticks than any of the inside cells inside

using double curly brackets to indicate that the code block should not be executed. The following code when used in a quarto document will render as shown in the example

````
````` markdown
This is **Quarto** code

```{{python}}
1 + 1
```
`````
````

qmd

8.2 Changing plot backgrounds

Plots and charts are useful in slides. Changing the background makes them fit in. This post will go over how to change the background of your plots to better match the slide background, in a handful of different libraries.

8.2.1 Why are we doing this?

If you are styling your slides to change the background color, you will find that most plotting libraries default to using a white background color. If your background is non-white it will stick out like a sore thumb. I find that changing the background color to something transparent `#FFFFFF00` is the easiest course of action.

Why make the background transparent instead of making it match the background?

It is simply easier that way. There is only one color we need to set and it is `#FFFFFF00`. This works even if the slide background color is different from slide to slide, or if the background is a non-solid color.

8.2.2 base R

we don't have to make any changes to the R code, we can supply the chunk options `dev` and `dev.args` for the chunk to `"png"` and `list(bg="transparent")` respectively and you are good. The chunk will look like this.

````
```{r, dev = "png", dev.args=list(bg="transparent")}
plot(mpg ~ disp, data = mtcars, col = factor(am), pch = 16, bty = "n")
```
````

You can also change the options globally using the following options in the yaml.

```yaml
knitr:
  opts_chunk:
    dev: png
    dev.args: { bg: "transparent" }
```



8.2.3 ggplot2

ggplot2 are handled the same way as base R plotting, so we don't have to make any changes to the R code, we can supply the chunk options `dev` and `dev.args` for the chunk to `"png"` and `list(bg="transparent")` respectively and you are good. The chunk will look like this.

````
```{r, dev = "png", dev.args=list(bg="transparent")}
library(ggplot2)
mtcars |>
  ggplot(aes(disp, mpg, color = factor(am))) +
  geom_point() +
  theme_minimal()
```
````

You can also change the options globally using the following options in the yaml.

```
knitr:
  opts_chunk:
    dev: png
    dev.args: { bg: "transparent" }
```



8.2.4 matplotlib

With matplotlib, we need to set the background color twice, once for the plotting area, and once for the area outside the plotting area.

```
fig = plt.figure()
# outside plotting area
plt.axes().set_facecolor("#FFFFFF00")
```

```
# your plot
plt.scatter(x, y, c=colors)

# sinde plotting area
fig.patch.set_facecolor("#FFFFFF00")
```



8.2.5 seaborn

For seaborn, we also set it twice, both of them in `set_style()`

```
sns.set_style(rc={'axes.facecolor':'#FFFFFF00',
                  'figure.facecolor':'#FFFFFF00'})
```



8.2.6 Source Document

The above was generated with this document.

source document

8.3 Plot sizing

Plots and charts are useful in slides. But we need to make sure they are sized correctly to be as effective as possible.

8.3.1 auto-stretch option

Revealjs slides default to having the option auto-stretch: true, this ensures that figures always fit inside the slide. You can turn it off globally like this.

```
format:
  revealjs:
    auto-stretch: false
```

or on a slide-by-slide basis by adding the `.nostretch` class to the slide.

```
## Slide Title {.nostretch}
```

We see how they affect sizing in the following slides first with the default, and second with `.nostretch`.



no auto-stretch



By themselves, they look pretty similar. One occasion where you really notice the difference is when there are other elements on the slide. `auto-stretch` makes sure the image fits by making the image smaller as seen below.

auto-stretch

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras lobortis leo a lacus viverra dictum. Nulla faucibus tortor a arcu tempus tempor. Aliquam tempus viverra blandit. Donec vitae accumsan tellus, rhoncus condimentum urna. Nunc vitae ultricies orci, et dictum sapien. Mauris iaculis blandit accumsan. Proin lobortis, nisi eget sollicitudin mattis, risus nisl laoreet augue, ut varius erat orci ut ante.



no auto-stretch

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras lobortis leo a lacus viverra dictum. Nulla faucibus tortor a arcu tempus tempor. Aliquam tempus viverra blandit. Donec vitae accumsan tellus, rhoncus condimentum urna. Nunc vitae ultricies orci, et dictum sapien. Mauris iaculis blandit accumsan. Proin lobortis, nisi eget sollicitudin mattis, risus nisl laoreet augue, ut varius erat orci ut ante.



8.3.2 Sizing Options

When sizing plots we need to remember that we have to deal with two kinds of sizes. First is the size of the actual file on disk, this is controlled using `out-width` and `out-height`. Next is how big the image is supposed to be in the document, which is controlled using `fig-width`, `fig-height`, and/or `fig-asp`. Lastly, you can control the location using `fig-align` and the resolution using `fig-dpi`.

All of these numbers will change depending on whether you have a title or other elements on your slides, what fonts you use, and the aspect ratio of the slides themselves.

8.3.2.1 out-width, out-height

Setting these options affects the size of the resulting image on disk. If they are set smaller than usual, we get an image that doesn't take up the whole screen.

````
```{r}
#| out-width: 6in
#| out-height: 3.5in
```
````

auto-stretch - `out-width: 6in` & `out-height: 3in`

auto-stretch - `out-width: 12in` & `out-height: 6in`

I don't find myself using these options much as I tend to want images that take up most of the space, but they are useful to know.

8.3.3 fig-width, fig-height

I end up using `fig-width` and `fig-height` the most out of the options shown in this blog post. I find that the default values are too high, making the text on the plot too small for the viewer to see. Especially for an in-person audience.

Below is the same chart 4 times with different value pairs for `fig-width` and `fig-height`. notice how the default values appear to be around `fig-width: 9` and `fig-height: 5`.

auto-stretch `fig-width: 12` & `fig-height: 8`

auto-stretch `fig-width: 9` & `fig-height: 5`



auto-stretch `fig-width: 6` & `fig-height: 3.5`



auto-stretch `fig-width: 3` & `fig-height: 1.6`



All of the above figures have roughly the same aspect ratios, but if you want others you just specify different values. Like this square chart below.

auto-stretch `fig-width: 5` & `fig-height: 5`



8.3.4 fig-asp

You might have noticed that the ratios shown in the last section weren't identical. Because unless you deal with 1-2 or 1-1 ratios you are going to get decimals very fast. And you have to recalculate small things over and over again. This is why `fig-asp` is amazing. Simply determine the aspect ratio between the height and width, set that as the `fig-asp` and then you just have to set one of `fig-height` or `fig-width`. Is it too small? increase `fig-height` and keep `fig-asp` the same. is it too big? decrease `fig-height` and keep `fig-asp` the same.

auto-stretch `fig-width: 6` & `fig-asp: 0.618`



auto-stretch `fig-width: 4` & `fig-asp: 0.618`



8.3.5 fig-align

Unless your chart fits fully inside the slide then it tends to be left aligned, you can change that with `fig-align`, setting it to `left`, `center` or `right`.

auto-stretch `fig-width: 4` & `fig-asp: 0.618` & `fig-align: left`



auto-stretch `fig-width: 4` & `fig-asp: 0.618` & `fig-align: center`



auto-stretch `fig-width: 4` & `fig-asp: 0.618` & `fig-align: right`



8.3.6 fig-dpi

Lastly, something you might need to worry about is the **D**ots **P**er **I**nch (DPI) specified by `fig-dpi`. This is a measure of resolution in your chart. If you see your chart becoming a little blurry, increase the dpi until it isn't anymore. Note that dpi will result in larger file sizes, so only change if you have to.

auto-stretch `fig-width: 3` & `fig-height: 1.6` & `fig-dpi: 50`



auto-stretch `fig-width: 3` & `fig-height: 1.6` & `fig-dpi: 300`



8.3.7 Make work with columns

even if you set the option globally, you will have to make slide-by-slide adjustments, such as with charts in `.columns`. Below is one example of how we can modify the `fig-asp` to make it look decent in a column layout.

auto-stretch `fig-width: 4` & `fig-asp: 1.236`



8.3.8 Source Document

The above was generated with this document.

source document

9 Layout

9.1 Pulling things left and right

Using multiple columns is a nice way to split up the content of your slides. I use it so much that I have a snippet to save time since I use it so much. It works great for side-by-side comparisons as well. This is done using the following syntax. We use the `width` attribute to determine the width of each of the columns.

```
:::: {.columns}

::: {.column width="40%"}
Left column
:::

::: {.column width="60%"}
Right column
:::

::::
```

but you can use it in quite a few ways beyond this! Firstly, each column is by itself a div, so you can style it directly. Such as making the right column have the right aligned text.

```
:::: {.columns}

::: {.column width="50%"}
Left column
:::

::: {.column width="50%" style="text-align: right;"}
Right column
:::

::::
```

The structure of columns doesn't require that we just use 2 columns. You can do as many columns as you want, but generally, you will have a hard time using more than 4.

```
:::: {.columns}

::: {.column width="25%"}
1st column
:::

::: {.column width="25%"}
2nd column
:::

::: {.column width="25%"}
3rd column
:::

::: {.column width="25%"}
4th column
:::

::::
```

Another way I like to use columns is by keeping one of them empty. This way provides a fast and easy way to add space or put text in specific locations.

```
:::: {.columns}

::: {.column width="30%"}
:::

::: {.column width="70%"}
Only right side
:::

::::
```

qmd

9.2 r-fit-text

Another quick way to change the layout of your slide is to let the text take up the entire slide real estate. We can do this using the r-fit-text class using the following syntax. This will make the text so big that it takes up all the horizontal space on the slide.

```
::: r-fit-text
Big Text
:::
```

This works well combined with the center class, which makes sure the text appears in the center vertically on the slide.

```
## {.center}

::: r-fit-text
This fits perfectly!
:::
```

One thing about using `r-fit-text` is that it applies the same text size to all the text inside it, so when you use it across multiple lines, you won't get the same effect.

```
::: r-fit-text
This fits perfectly!

On two lines
:::
```

This can however be fixed, by using a `r-fit-text` for each line of text.

```
::: r-fit-text
This fits perfectly!
:::

::: r-fit-text
On two lines
:::
```

qmd

9.3 Using images

Using images for style is another thing you can do to change the layout. If you don't have much content on your slides. e.i. just a sentence or two. You could pair that with an image that relays some of the same information. These images will typically not contain content themselves but rather reinforce the content on the slides.

There are 3 main ways to include images. Using the basics, using absolute position or as background images.

9.3.1 Basic figures

The basic way of adding figures is using the following syntax

```
![](holly-mandarich.jpg)
```

where `holly-mandarich.jpg` is the path to the image. We can change some options such as adding `{fig-align="right"}` to the end to change the alignment. But I end up not using this style that much because it is added as content, so it will push other content around, and it adheres to margins which I rarely want for tasks like this.

Photo by Holly Mandarich on Unsplash

qmd

9.3.2 Absolute position

Absolute is my favorite way of adding images. It gives me much more control over where the image is located and its size.

You use the following syntax:

```
![](noelle-rebekah.jpg){.absolute top=0 right=0 height="100%"}
```

Where you can use the following attributes:

Attribute	Description
`width`	Width of element
`height`	Height of element
`top`	Distance from top of slide
`left`	Distance from left of slide

Attribute	Description
`bottom`	Distance from bottom of slide
`right`	Distance from right of slide

you need one of `left` and `right` and one of `bottom` and `top` to give the correct location. I find that just using one of `width` or `height` is easier as it doesn't distort the image. All of these attributes accept all known CSS values, such as pixels, inches, and percentages. All about CSS length for more information.

All images in revealjs default to the following maximum sizes:

```
max-width: 95%;
max-height: 95%;
```

No matter how large we set `width` or `height` we are overruled by `max-width` and `max-height`. We can make the image any size by overruling those. Specifically, we can unset them with `style="max-height: unset; max-width: unset;"`.

With some experimentation, we can size the image such that it is where we want it. Notice that we are using negative locations to make this happen as `0` is inside the slide.

```
![](noelle-rebekah.jpg){.absolute top="-10%" right="-10%" height="120%" style="max-height: u
```

Photo by Noelle Rebekah on Unsplash

qmd

> **Warning**
>
> Since the way that positions are done in revealjs, it can be almost impossible to have the above effect for all aspect ratios. Make it work for the aspect ratio you use, and have peace.

9.4 Absolute position everything

It took me too long to realize, but the absolute class can be used on anything!

You might have seen the following syntax to place an image anywhere

```
![](image1.png){.absolute top=200 left=0 width="350" height="300"}
```

But you are not restricted in using it with images. The following results in the image you see attached:

I like to use it with text in the following way:

```
[python is great]{.absolute bottom="45%" left="20%"}

[and so is R]{.absolute bottom="0%" right="0%"}
```

qmd

9.4.1 Background image

The last way to add images, which I highly recommend is the use of background images. And in many ways, it is the simplest one.

you specify it on the slide level in the following way:

```
## Slide Title {background-image="galen-crout.jpg"}
```

you can set other options such as background-position and background-repeat but I rarely end up using them.

I end up not setting a default title and use `.absolute` to place any content I want where I want it.

```
## {background-image="galen-crout.jpg"}

[always explore]{.absolute left="50%" top="20%" style="rotate: -10deg;"}
```

Photo by Galen Crout on Unsplash

qmd

As we see here, the text positioning can change how the slides are perceived. Both in style and emotion, try to think about how you can incorporate text positioning to maximize engagement.

9.5 Overlayed Text Boxes

When using background images, it can be hard to place text on top of it, in a way that that keeps the text readable. This is often an issue with images that are more busy or have colors that match. A simple fix is to overlay a box, and we then add text on you. If we take the first slide here as an example.

```
## {background-image="tim-marshall.jpg"}
```

First, we try to use `.absolute` to add some inspiring text. It adds text, but it is not easy to read at all!!

```
## {background-image="tim-marshall.jpg"}

::: {.absolute left="55%" top="55%" style="font-size:1.8em;"}
Be Brave

Take Risks
:::
```

But we can expand on this idea, adding a `background-color` to make it stand out. We also added some `padding`, otherwise the background would just be slightly bigger than the text itself.

9.6

```
::: {.absolute left="55%" top="55%" style="font-size:1.8em; padding: 0.5em 1em; background-c
Be Brave

Take Risks
:::
```

it already looks quite good! We can make it pop just a little bit more, by adding a `backdrop-filter` to make it look a little glass-like, adding a `box-shadow` to make it look a little 3-dimensional, and adding a small `border-radius` to stop the sharp corners.

```
## {background-image="tim-marshall.jpg"}

::: {.absolute left="55%" top="55%" style="font-size:1.8em; padding: 0.5em 1em; background-c
Be Brave

Take Risks
:::
```

Photo by Tim Marshall on Unsplash

qmd

I think this turned out well. There are endless ways to use this. It is quite CSS-heavy work, but I think it is worth it. Note that you are always free to copy an example and modify it to your wants.

9.7 Vary the type of slides

This post has shown a number of ways to place content on your slides. My final advice in this post is to use some of these tips to vary how the content is laid out. It doesn't have to be over the top, but slight variations can help a slide deck feel fresh.

10 Manual code

10.1 Hand-styled code chunks

With a little extra effort, you can create highlighted code chunks. This is different than the code highlighting that comes naturally. In this instance, we are creating an unstyled code chunk, and styling part of the code manually with css classes.

First, we write a new css class, I like to call it `.mono`. The important thing is that we set `font-family: monospace;`, but we can do other things, like setting the `font-size`.

```
.mono {
  font-family: monospace;
  font-size: 0.9em;
}
```

Next, we add our code in a fenced div, with the mono class. you need to use \ to add leading spaces, and each line ends with 2 spaces to denote newlines

```
::: mono
library(ggplot2)
mtcars |>
\ \ ggplot(aes(mpg, disp)) +
\ \ geom_point() +
\ \ geom_smooth(method = "lm", formula = "y ~ x")
:::
```

Taking it to the next step, you can manually change the formatting of functions using css or css classes, `library([ggplot2]{style="color:purple;"})` would make `ggplot2` purple. This is also a great place to use css classes.

We can take it a step further and use fragments to have the code highlighted one bit at a time. Changing the last line to the following

```
geom_smooth([method = "lm"]{.fragment .highlight-red}, [formula = "y ~ x"]{.fragment .highli
```

seperately highlights the code `method = "lm"` then `formula = "y ~ x"` in red and blue.

qmd

11 asciicast

This chapter celebrates the asciicast R package to showcase rich output.

11.1 Without asciicast

When R code produces messages with colors or styling, they are typically lost when we write slides

qmd

11.2 Default asciicast

This is where the amazing asciicast R package comes into play. {asciicast} takes an R script and turns it into an asciinema cast. We will use the very convenient init_knitr_engine() function to make this as easy as possible.

We can add the {asciicast} {knitr} engine by placing the following chunk at the beginning of our document.

````
```{r}
asciicast::init_knitr_engine()
```
````

And then we toggle asciicast output by replacing

````
```{r}
library(tidymodels)
```
````

with

````
```{asciicast}
library(tidymodels)
```
````

With this change, we get the following output, nicely styled with colors and everything.

qmd

> **Note**
>
> The output of asciicast will by default fill the screen horizontally, but for some reason when setting `format: revealjs: self-contained: true` it doesn't. All examples use this option for a better blogging experience. You just have to trust me that it looks better without the option.

11.3 Using startup to seed, load some packages

The full list of arguments to `asciicast_init_knitr_engine()` is as follows

```
formals(asciicast::init_knitr_engine)
```

```
$echo
[1] FALSE

$same_process
[1] TRUE

$timeout
[1] 10

$startup
NULL

$record_env
NULL

$echo_input
[1] TRUE

$interactive
[1] TRUE
```

I will talk about the 3 I use, which are `same_process`, `echo`, and `echo_input`. The two go together. `same_process` which defaults to `TRUE`, does what it says. It determines whether all

the asciicast chunks should be in the same process or not. If you want to show things that only happen once a session, you might want to turn this off which is done this way

````
```{r}
asciicast::init_knitr_engine(
  same_process = FALSE
)
```
````

and gives the following results

qmd

You will notice that the code that is being run is included in the asciicast output. This is happening because the default arguments `echo = FALSE` and `echo_input = TRUE`. By swapping these options we get the code as normal quarto code, with the output as asciicast

````
```{r}
asciicast::init_knitr_engine(
  echo = TRUE,
  echo_input = FALSE
)
```
````

qmd

It wouldn't make much sense to set both of these to `TRUE` as you would have duplication, but there are certainly cases where you want both of these to be `FALSE` as you just want the output.

11.4 Using theme argument

The output is styled a certain way, and we can change it. At the time of writing, the following 10 premade themes are available.

- asciinema
- tango
- solarized-dark
- solarized-light
- seti
- monokai
- github-light

- github-dark
- pkgdown
- readme

We can toggle any of these by setting the option `asciicast_theme`. The below example showcases the `"solarized-light"` theme.

````
```{r}
options(asciicast_theme = "solarized-light")
```
````

qmd

If none of these is what you need, you can use a completely custom theme. I suggest that you modify one of the existing theme

For this example, the `github-light` theme was modified by changing the `background` to pure white to match the background of the slide, thus having it blend into the slide.

````
```{r}
options(
  asciicast_theme = list(
    black         = c(grDevices::col2rgb("#073642")),
    red           = c(grDevices::col2rgb("#dc322f")),
    green         = c(grDevices::col2rgb("#859900")),
    yellow        = c(grDevices::col2rgb("#b58900")),
    blue          = c(grDevices::col2rgb("#268bd2")),
    magenta       = c(grDevices::col2rgb("#d33682")),
    cyan          = c(grDevices::col2rgb("#2aa198")),
    white         = c(grDevices::col2rgb("#eee8d5")),
    light_black   = c(grDevices::col2rgb("#002b36")),
    light_red     = c(grDevices::col2rgb("#cb4b16")),
    light_green   = c(grDevices::col2rgb("#586e75")),
    light_yellow  = c(grDevices::col2rgb("#657c83")),
    light_blue    = c(grDevices::col2rgb("#839496")),
    light_magenta = c(grDevices::col2rgb("#6c71c4")),
    light_cyan    = c(grDevices::col2rgb("#93a1a1")),
    light_white   = c(grDevices::col2rgb("#fdf6e3")),
    background    = c(grDevices::col2rgb("#ffffff")),
    cursor        = c(grDevices::col2rgb("#657b83")),
    bold          = c(grDevices::col2rgb("#657b83")),
    text          = c(grDevices::col2rgb("#657b83"))
  )
```
````

```
)
```

qmd

11.5 Roundup

I just recently learned about {asciicast} and I love it. If you know of any other cool tools or packages, please reach out and share them with me!

Part III

Interactivity

12 Fragments

12.1 Highlight incremental slides

The use of incremental lists is a great way to add a little something to a set of slides. It also avoids a wall of text, allowing the presenter to show one bullet at a time. All in all, this is helpful as it can be used to focus the viewers' attention.

As a reminder, we create an incremental list using the following syntax:

```
::: {.incremental}
- thing 1
- thing 2
:::
```

We can add another class to this div and use it to style it more.

```
::: {.incremental .highlight-last}
- thing 1
- thing 2
:::
```

then we use this to style our list. Below `.current-fragment` refers to the last shown item in the list. Setting the `color: grey` isn't necessary, but it is a way to downplay the "not-current" items

```
.highlight-last  {
  color: grey;
  .current-fragment {
    color: #5500ff;
  }
}
```

These together yield these slides:

qmd scss

12.2 Changing fragments with CSS

At the most fundamental level, a fragment can be split into 3 stages

- before
- current
- after

determining which stage is handled completely by revealjs using the `fragment-index` attribute. The way we can make things happen is to apply a different style to each of the 3 stages.

the maximal fragment signature is as follows, with `fragment-name` being the name of the fragment in question. For them to work properly you have to list them in the following order. Which corresponds to `before`, `after`, and `current`.

```
.reveal .slides section .fragment.fragment-name {
}

.reveal .slides section .fragment.fragment-name.visible {
}

.reveal .slides section .fragment.fragment-name.current-fragment {
}
```

> **!** Important
>
> The reason why this ordering is important is because `.visible` and `.current-fragment` are triggered at the same time. And because I simplified the order a little too much. There isn't `before`, `current`, and `after`. Instead, we have `always`, `current`, and `not-before-current`. In essence, they do the same, as long as you order them in this order to make sure they cascade properly.

Before we try to implement a fragment by ourselves, we need to note one thing real quick. Each of these stages is styled a specific way by default. In practice, what this means is that the `before` style has the following attributes set to make the text invisible:

```
opacity: 0;
visibility: hidden;
```

If you want the text to be visible before the fragment triggers, simply set these two attributes to `unset`.

Another note I would like to add is that while you are able to modify anything in a fragment, as it is just triggering CSS, you should be careful about position and size. While you might be able to make it work, it is likely to cause a lot of shifting and jittering as elements resize.

> Tip
>
> Looking at the source code for the default fragments gives us a good idea for how different styles of fragments work.

12.3 Example 1

This first example illustrates how the different phases work in a fragment. We have thus created an `rgb` fragment that assigns a different color to each of the 3 phases. We `unset` both `opacity` and `visibility` to have the text appear beforehand. This leaves us with the following fragment:

```
.reveal .slides section .fragment.rgb {
  opacity: unset;
  visibility: unset;
  color: red;
}

.reveal .slides section .fragment.rgb.visible {
  color: blue;
}

.reveal .slides section .fragment.rgb.current-fragment {
  color: green;
}
```

Advancing and de-advancing(?) the slides showcase how the different classes are applied for fragments.

qmd scss

Worth noting that this single fragment could be rewritten as the following using SCSS nesting.

```
.reveal .slides section .fragment.rgb {
  opacity: unset;
  visibility: unset;
```

```
  color: red;

  &.visible {
    color: blue;
  }

  &.current-fragment {
    color: green;
  }
}
```

12.4 Example 2

One custom fragment I use from time to time is the background highlighted style. And it is very simple, instead of changing the color of the text, it changes the background color. I find that it is a much stronger indication than changing the text itself.

This fragment gives us two things. I leave the text visible beforehand. Then it turns the background orange, and after it lightens the background color a little bit.

```
$theme-orange: #FFB81A;

.reveal .slides section .fragment.hl-orange {
  opacity: unset;
  visibility: unset;

  &.visible {
    background-color: $theme-orange;
  }

  &.current-fragment {
    background-color: darken($theme-orange, 10%);
  }
}
```

This once is nice and flexible because it is easy to extend.

```
$theme-orange: #FFB81A;
$theme-yellow: #FFD571;
$theme-brown: #E2AE86;
$theme-pink: #FED7E1;
```

```
.reveal .slides section .fragment {

  &.hl-orange,
  &.hl-yellow,
  &.hl-pink,
  &.hl-brown {
    opacity: 1;
    visibility: inherit
  }

  &.hl-brown.visible {
    background-color: $theme-brown;
  }

  &.hl-brown.current-fragment {
    background-color: darken($theme-brown, 10%);
  }

  &.hl-orange.visible {
    background-color: $theme-orange;
  }

  &.hl-orange.current-fragment {
    background-color: darken($theme-orange, 10%);
  }

  &.hl-yellow.visible {
    background-color: $theme-yellow;
  }

  &.hl-yellow.current-fragment {
    background-color: darken($theme-yellow, 10%);
  }

  &.hl-pink.visible {
    background-color: $theme-pink;
  }

  &.hl-pink.current-fragment {
    background-color: darken($theme-pink, 10%);
  }
}
```

And we are willing to tap into some scss we can condense it down quite a lot using SCSS loops.

```
$colors: (
  "orange": #FFB81A,
  "yellow": #FFD571,
  "brown": #E2AE86,
  "pink": #FED7E1
);

@each $name, $color in $colors {
  .reveal .slides section .fragment.hl-#{$name} {
    opacity: unset;
    visibility: unset;

    &.visible {
      background-color: lighten($color, 5%);
    }

    &.current-fragment {
      background-color: $color;
    }
  }
}
```

qmd scss

12.5 Example 3

The last example included a bit of flair by having the current fragment element be a slightly different color and then changing it after. We can simplify it a bit by not specifying the `.current-fragment` class.

```
$theme-orange: #FFB81A;

.reveal .slides section .fragment.hl-orange {
  opacity: unset;
  visibility: unset;
```

```
  &.visible {
    background-color: $theme-orange;
  }
}
```

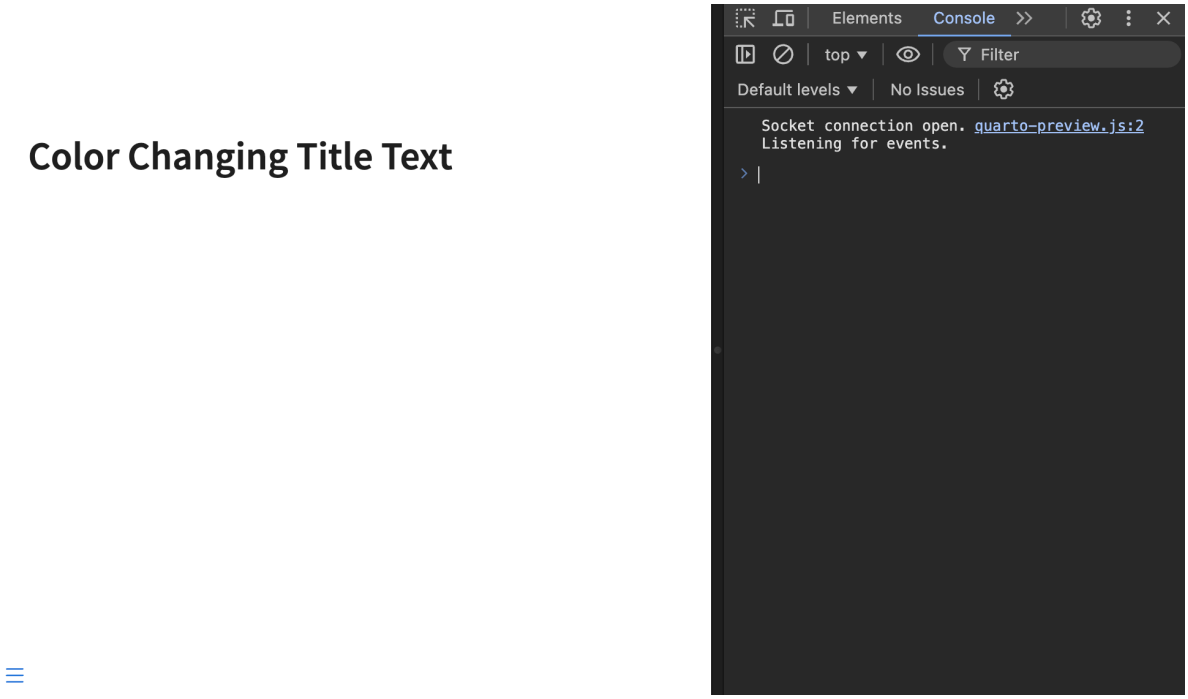
This fragment works more or less the same way as before but doesn't change color once it is applied. It will be a more appropriate fragment many times.

This leads us to our final piece of knowledge in this blog post. We don't have to fully specify a fragment. We just have to declare how we want it to behave differently, and then the default "stay hidden, then appear" fragment.

12.6 Fragments 201

When a fragment is either shown or hidden `reveal.js` (the engine that powers our slides) will dispatch an event. This event can be picked up using JavaScript.

You will need a little bit of Javascript knowledge, but I found that you don't need a lot of knowledge to produce useful things for slides. Once your slides are rendered in your browser, you can toggle the developer tools, where you can find a javascript console. This is where I do the work needed.



We can capture the event using the following snippets of code

```
Reveal.on('fragmentshown', (event) => {
  // event.fragment = the fragment DOM element
});
Reveal.on('fragmenthidden', (event) => {
  // event.fragment = the fragment DOM element
});
```

`Reveal` is the javascript object that powers the whole presentation. To have fun things happening when we use fragments, we need to write some code inside these curly brackets. The first chunk of code runs whenever a fragment appears, and the second runs whenever a fragment disappears. In in this environment, we have access to the `event` which is the DOM element of fragment div itself as created in our slides. We can take advantage of that in different ways as you will see.

```
Event {
  "isTrusted": false,
  "fragment": "Node",
  "fragments": ["Node"],
  "type": "fragmentshown",
  "target": "Node",
  "currentTarget": "Node",
  "eventPhase": 2,
  "bubbles": true,
  "cancelable": true,
  "defaultPrevented": false,
  "composed": false,
  "timeStamp": 2259.5,
  "srcElement": "Node",
  "returnValue": true,
  "cancelBubble": false,
  "NONE": 0,
  "CAPTURING_PHASE": 1,
  "AT_TARGET": 2,
  "BUBBLING_PHASE": 3
}
```

Last note, you can have multiple of these `Reveal.on()` statements, as they will trigger one after another. So depending on how you want to organize, both styles are valid.

```
// one statement
Reveal.on('fragmentshown', (event) => {
  fragment_style_1(event);
  fragment_style_2(event);
  fragment_style_3(event);
});

// multiple statements
Reveal.on('fragmentshown', (event) => {
  fragment_style_1(event);
});
Reveal.on('fragmentshown', (event) => {
  fragment_style_2(event);
});
Reveal.on('fragmentshown', (event) => {
  fragment_style_3(event);
});
```

Lastly, the way I set up my revealjs slides to do javascript is by setting the `include-after-body` attribute in the yaml file,

```
format:
  revealjs:
    include-after-body:
      - "_color.html"
```

and having it point to a file that looks like this:

```
<script type="text/javascript">

</script>
```

then inside we put my javascript code, which for this blog post will be some `Reveal.on()` calls.

12.7 Color changing

This first example is going to be an illustrative example of what we can do and how to do it. And it will thus not be very useful.

The first lesson I want to show is that you are not limited to modifying the content inside the fragment. This means that we can actually have empty fragments that modify some other element. So in our document, we could have a slide that looks like this:

```
## Color Changing Title Text

::: {.fragment .color}
:::
```

I want to change the color of the header when the fragment triggers. To do that we need two things.

1. The color to change it into
2. Access to the header element

The first part is easy, I found a "random javascript" script online. We start by assigning that to a variable.

```
random_color = '#'+(Math.random()*0xFFFFFF<<0).toString(16);
```

Next, we need to find the header. Remember the `Reveal` object I mentioned earlier? It has a very handy `.getCurrentSlide()` method. When run we get the current slide we are on, which is exactly what we need.

```
Reveal.getCurrentSlide()
<section id="color-changing-title-text" class="slide level2 present" style="display: block;"
  <h2>Color Changing Title Text</h2>
  <div class="fragment color" data-fragment-index="0"></div>
  <div class="quarto-auto-generated-content"></div>
</section>
```

From this, we can get to the title using `.querySelector()`

> **Note**
>
> We don't need `.querySelectorAll()` because by definition there will only be one `h2` on a quarto slide using default options.

```
Reveal
  .getCurrentSlide()
  .querySelector("h2")
<h2>Color Changing Title Text</h2>
```

We can then change the color by selecting the `style` element of the div and updating the `color` variable.

```
Reveal
  .getCurrentSlide()
  .querySelector("h2")
  .style
  .color = random_color;
```

And that is technically all we need. Put that code inside the `Reveal.on()` statements, and the color of the header will change each time the fragment is triggered.

One thing worth remembering is that this javascript code will run everything a fragment is run. So to limit it, we can make sure it only runs when we want it to. This is why I gave the fragment a `.color` class. We can use the following `if` statement to make sure our code only runs when we want it to.

```
if (event.fragment.classList.contains("color")) {

}
```

We could stop here. But I want to show a little more with this example. For right the color changes randomly, but we could allow for a little bit of information transfer. HTML has this concept called datasets. Each div can have a data set of information. We should use this to give our fragments more flexibility.

Luckily it is quite effortless to specify data set values in quarto. Below is the same fragment div as before, but with a data set value named `color`.

```
:::: {.fragment .color data-color="orange"}
:::
```

We can now on the javascript side pull out this value with ease.

```
color = event.fragment.dataset.color;
```

> **Warning**
>
> We are not doing any input checking, so this code will fail silently if you don't have a color specified in the div.

And set it the same as before.

```
Reveal
  .getCurrentSlide()
  .querySelector("h2")
  .style
  .color = color;
```

This will give us the final fragment code as follows

```
Reveal.on('fragmentshown', (event) => {
  if (event.fragment.classList.contains("color")) {
 random_color = '#'+(Math.random()*0xFFFFFF<<0).toString(16);

 Reveal
      .getCurrentSlide()
      .querySelector("h2")
      .style
      .color = random_color;
  }
});

Reveal.on('fragmenthidden', (event) => {
  if (event.fragment.classList.contains("color")) {
 color = event.fragment.dataset.color;

 Reveal
      .getCurrentSlide()
      .querySelector("h2")
      .style
      .color = color;
  }
});
```

qmd js

12.8 Scroll output

Sometimes you run into a situation where you want to interact with an element on a slide. This can happen when you need to scroll or toggle something. While that would be fine to do by hand, it can be hard to do casually, and impossible to do if you are using a clicker.

Scrolling text in a window is one thing that isn't that hard to do with JavaScript.

We will follow the same steps as before.

1. Find the element we want to show
2. Figure out how to scroll it

The element can again be found using `.getCurrentSlide()` and `querySelector()` after a little digging.

```
Reveal
  .getCurrentSlide()
  .querySelector(".cell-output code")
```

Next, we need to figure out how to scroll it. This can be done using the .scrollTo() method. This function should be passed on to how much we want to scroll and how. As far as I know, this can only be set using pixel values so we have to try a couple of times to get it right. `1000` appears enough for this example to get us all the way to the bottom. Setting `behavior` to smooth for a little flair.

```
{
  top: 1000,
  behavior: "smooth",
}
```

This means that the fragment is finished with

```
Reveal.on('fragmentshown', (event) => {
  if (event.fragment.classList.contains("scroll")) {
 Reveal
    .getCurrentSlide()
    .querySelector(".cell-output code")
    .scrollTo({
 top: 1000,
 behavior: "smooth",
    })
  }
});
```

But wait! What if you have to go back? this is where `fragmenthidden` is needed, we simply take the preview code and say we want to go back to the top by setting `top` to `0`.

```
Reveal.on('fragmenthidden', (event) => {
  if (event.fragment.classList.contains("scroll")) {
 Reveal
    .getCurrentSlide()
    .querySelector(".cell-output code")
    .scrollTo({
 top: 0,
 behavior: "smooth",
    })
  }
});
```

> **Note**
>
> Some changes to our slides are really hard to reverse. They would thus make for bad fragments. You could implement them halfway without the `fragmenthidden` and you would just need to be really confident that you never have to go backwards in your slides.

qmd js

> **Tip**
>
> We didn't do it here, but you could use dataset values to help determine which elements should be scrolled and how much to scroll them by instead of hardcoding it all as we do here.

12.9 Tabset advance

Quarto also has tabset support for slides, which is again a very nice feature. It runs into the same clicker interaction we noted earlier. It requires a mouse to correctly toggle in the middle of a presentation.

We can deal with this as well. As always we need to find the elements and how to toggle them.

> **Note**
>
> The astute reader will notice that the following will only work on a tabset with 2 tabs. Making this fragment work for multiple tabs is left as an exercise for the reader.

We are again using `.getCurrentSlide()` and `querySelector()`, and with some trial and error, determine that the following two CSS selectors captures the two tabs.

- `.panel-tabset ul li:first-of-type a`
- `.panel-tabset ul li:last-of-type a`

And we are lucky because these elements have a working `.click()` method that we can use.

This means that the full fragment looks like this:

```
Reveal.on('fragmentshown', (event) => {
  if (event.fragment.classList.contains("tabswitch")) {
 Reveal
      .getCurrentSlide()
      .querySelector(".panel-tabset ul li:last-of-type a")
      .click()
  }
});

Reveal.on('fragmenthidden', (event) => {
  if (event.fragment.classList.contains("tabswitch")) {
 Reveal
      .getCurrentSlide()
      .querySelector(".panel-tabset ul li:first-of-type a")
      .click()
  }
});
```

qmd js

12.10 advance embedded slides

The last example I'll show for now is one you have seen me use already. I like to put quarto slides inside quarto slides. However, it becomes messy to advance the embedded slides, because they take focus of the mouse. I have used a fragment to advance these.

We start by embedding a set of slides in our set of slides. We do thing with `<iframe class="slide-deck" src="fragment-scroll.html" style="width:100%; height: 500px;" ></iframe>`.

The `Reveal` object has a fairly extensive API you can use. So we just need to fetch the right `Reveal` object so we can use the `.left()` and `.right()` methods to advance the slides. It took

me a while to find the right code, but `.contentWindow` was the missing piece. The following returns the embedded `Reveal` object.

```
Reveal
  .getCurrentSlide()
  .querySelector("iframe")
  .contentWindow
  .Reveal
```

Which then gives us the following as our fragment

```
Reveal.on('fragmentshown', event => {
  if (event.fragment.classList.contains("advance-slide")) {
 Reveal
      .getCurrentSlide()
       .querySelector("iframe")
      .contentWindow
      .Reveal
      .right()
    }
});
Reveal.on('fragmenthidden', event => {
  if (event.fragment.classList.contains("advance-slide")) {
 Reveal
      .getCurrentSlide()
      .querySelector("iframe")
      .contentWindow
      .Reveal
      .left()
    }
});
```

qmd js

Part IV

Extensions

13 letterbox

14 miscellaneous

14.1 Hiding slides

changing the slide visibility is as simple as setting `visibility="hidden"` attribute to the header of a slide

```
## Slide Title {visibility="hidden"}
```

I find this useful when I have to give the same presentation multiple times, and I have a disclaimer or other seasonally important slides. Instead of removing and reinserting the information each time, I just changed the attribute.

qmd

14.2 Avoid duplication using Includes

This last tip doesn't come with an example, as it doesn't get useful before you start working with multiple files. We are talking about the includes short code.

Using the following short code; `{{< include _content.qmd >}}` includes the content of `_content.qmd` into the document in a "copy-paste" manner before the rendering of the document.

This has proved useful for me when I want the same slides to appear at the start or end of multiple decks. And you are not limited to .qmd files! you can embed html files or svg too.

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.