Functionally building your book

The most obvious solution to me seemed to simply change what my code was doing depending on the format I was exporting to. I can tweak the final PDF output to be perfect, but the docx format has to be just good enough to get the point across to whoever I'm seeking feedback from.

You can set Bookdown to export to multiple formats, and set options for each export format, using an `_output.yml` file. But, weirdly, I don't actually find it easy to change code execution in your document based on what is currently being rendered by Bookdown. You'd think there'd be a way to declare `is(pdf)` or `is(docx)` and execute appropriately, right? Unfortunately not. [2]

According to this Stackoverflow thread, you can determine the currently rendering format with `rmarkdown::all_output_formats(knitr::current_input())[1]`. I *think* `_output.yml` is supposed to be 'shuffled' by Rstudio so that index 1 is always the *currently* rendering output, but that wasn't occurring when I tested it; every time I would just get back is the *first* format listed in `_output.yml`, even if it wasn't the one currently being rendered.So if `_output.yml` had `bookdown::pdf_document` and then `bookdown::word_document` in it, all I would get back is `bookdown::pdf_document` even when rendering to docx. [3]

I could just swap in an out various outputs as I need them, of course, but I ended up taking a step back at this point and instead considering what was going on when rendering my source files. If you're like me, you're using Rstudio as your IDE, and this gives you a handy 'knit' or 'build' button in your toolbar when it detects you're in a Rmarkdown/Bookdown project. While nice, this abstraction actually ended up masking for me the underlying method. Really all these things are doing is calling `bookdown::render_book`, itself a wrap around `rmarkdown::render`. This function takes an argument for the format(s) to render when called, using the options found in `_output.yml` for each particular output format.

Once I realised that rendering a book is just another function, I decided to just wrap it in my *own* function. I can then provide different, *unique* `_output.yml` files to each call to `bookdown::render_book` based on an argument I provide, using `yaml` and `ymlthis` packages to read and write the yaml files as I need them:

```
build_book <- function(format = "all"){

  switch(format,
    "all" = formats <- c("bookdown::pdf_document2",
                         "bookdown::word_document2"),
    "pdf" = formats <- "bookdown::pdf_document2",
    "word" = formats <- "bookdown::word_document2"
  )

  for(fmt in formats) {

    if(grepl("pdf", fmt)) {
      out_yml <- yaml::read_yaml("_pdf_output.yml")
      ymlthis::use_yml_file(ymlthis::as_yml(out_yml), "_output.yml")
    }
```

```
    if(grepl("word", fmt)) {
      out_yml <- yaml::read_yaml("_word_output.yml")
      ymlthis::use_yml_file(ymlthis::as_yml(out_yml), "_output.yml")
    }

    bookdown::render_book(here::here("index.Rmd"),
                          output_format = fmt)

    fs::file_delete("_output.yml")
  }
}
```

This pulls in specific yaml files from a project folder, and sets it as the `_output.yml` for Bookdown to find. Now, when rendering, `rmarkdown::all_output_formats(knitr::current_input())[1]` returns the *correct* file format for the current render, as it's the *only* format in the output yaml file! This allowed me to put this at the start of my bookdown source files:

```
fmt <- rmarkdown::all_output_formats(knitr::current_input())[1]
```

And then test for output with:

```
if(!grepl("word", fmt)) {
  my_table %>%
    kableExtra::kbl(caption = "My full table caption",
                    caption.short = "Short caption for ToC",
                    booktabs = TRUE) %>%
    kableExtra::kable_styling(latex_options = c("striped",
                                                "HOLD_position",
                                                "scale_down"))
} else {
  exTab %>%
    knitr::kable(caption = "My full table caption",
                 booktabs = TRUE)
```

This lets me, for example here, use kableExtra to further style my table for PDF/HTML output, but since kableExtra doesn't really work for docx, just use regular ol' `knitr::kable` to render it there!

Taking it further

Once I started writing my own book rendering wrapper function, the opportunities opened up in front of me. For example – one thing I like to do is build my drafts and save them into folders with today's date, so I can put my feedback from lecturers in that same folder and keep it all together for easy reference. So I added the following to my `build_book()` function:

```
build_path <- glue::glue("book/builds/{Sys.Date()}")

bookdown_yml <-
      ymlthis::yml_bookdown_opts(.yml = ymlthis::yml_empty(),
        book_filename = "My Project",
        rmd_subdir = c("src"),
        delete_merged_file = FALSE,
```

```
    output_dir = build_path
  )
```

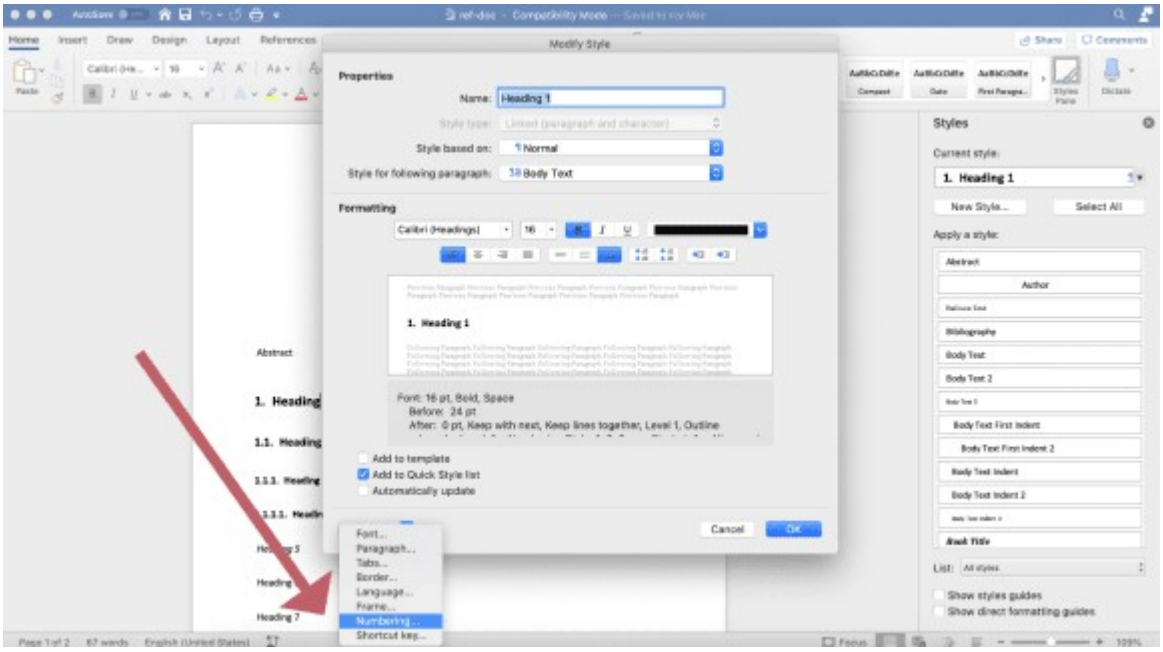
Now by creating a custom `_bookdown.yml` file for each call to `build_book()`, each build will save into a `book/src/DATE` specific folder!

There are now *tons* of ways to further change how Bookdown builds your book based on arguments you provide to your own `build_book`, swapping in and out various Bookdown configuration files as required. Is this the most efficient way to customise your Bookdown project? Probably not, with all the creating and changing of existing yaml files, but speed isn't really my concern here – running `build_book` and having to step back and wait for a minute or two for my code to run is much more preferable to the *stupid* amounts of time I spent correcting fiddly tables/plot/cross reference output by hand in Word before!

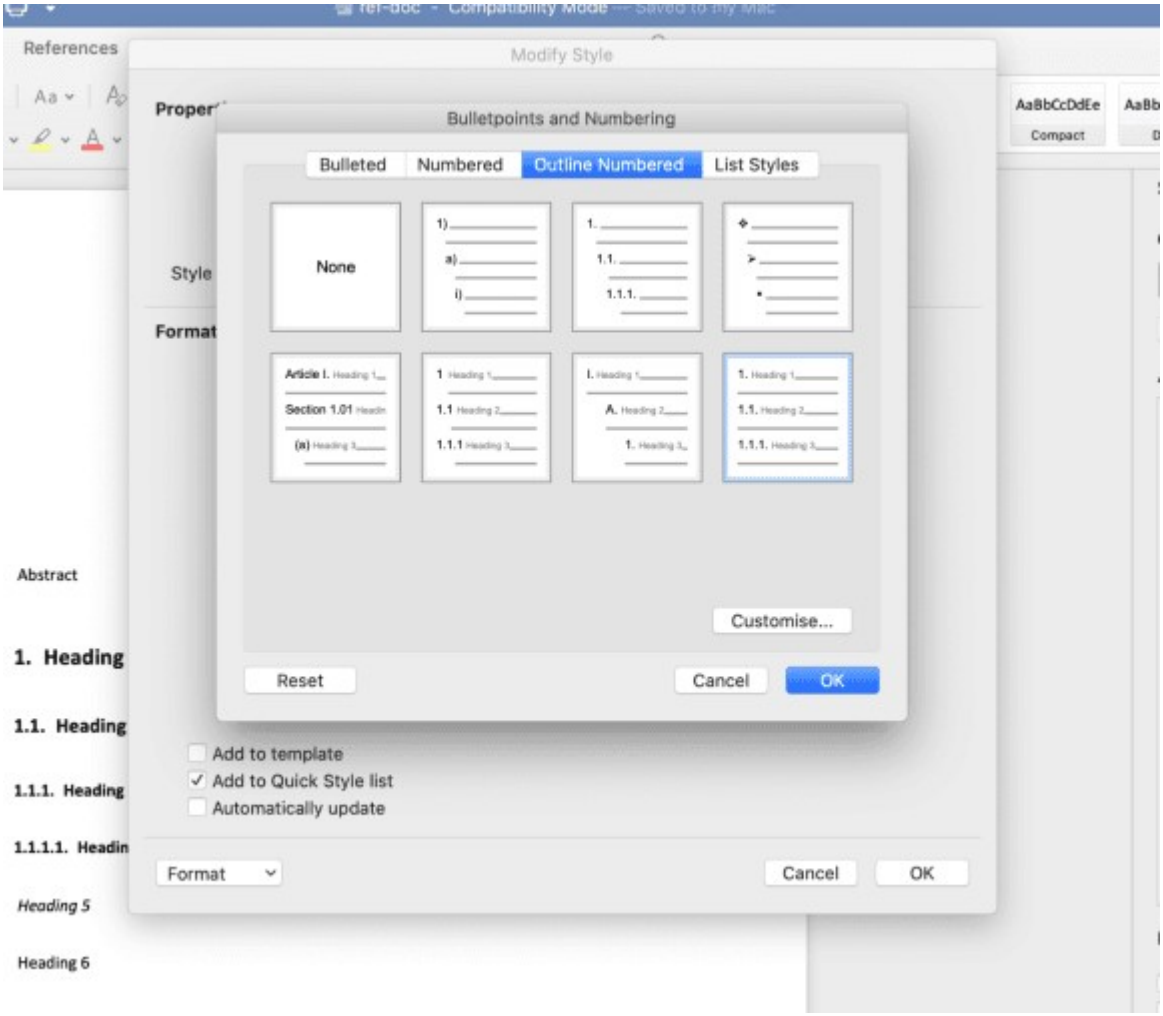
Word / .docx output tweaks

So now I can customise my output based on where I'm rendering, I need to fix a couple of other things in my docx output. These now need to be done by editing my reference document for Bookdown docx output. For a primer on creating and editing your reference doc, check out this article.

One extra thing I needed was *numbered* section headings, not just H1, H2 etc rendered as headings. This can't just be set in Bookdown, but instead can be done in your reference doc by selecting your header, going to `Format -> Style... -> Modify`, and then finding this hidden gem of a menu in the lower left of the style formatting menu:



Then clicking `Outline Numbered`:



You can also click ‘customise’ here for some extra options, such as how much sub-headings should indent. Be aware you also need to do this for *every* heading you want numbered, so H1, H2, etc. I then combined this with `build_book` to choose whether to number headings in docx output:

```
build_book <- function(format = "word", word_num = TRUE){

    switch(format,
    "all" = formats <- c("bookdown::pdf_document2",
                         "bookdown::word_document2"),
    "pdf" = formats <- "bookdown::pdf_document2",
    "word" = formats <- "bookdown::word_document2"
  )

  for(fmt in formats) {

    if(grepl("pdf", fmt)) {
      out_yml <- yaml::read_yaml("_pdf_output.yml")
      ymlthis::use_yml_file(ymlthis::as_yml(out_yml), "_output.yml")
    }

    if(grepl("word", fmt)) {
      if(word_num = TRUE) {
        out_yml <- yaml::read_yaml("_word_output_numbered.yml")
        ymlthis::use_yml_file(ymlthis::as_yml(out_yml), "_output.yml")
```

```
        } else {
        out_yml <- yaml::read_yaml("_word_output.yml")
        ymlthis::use_yml_file(ymlthis::as_yml(out_yml), "_output.yml")
        }
    }
 }

    bookdown::render_book(here::here("index.Rmd"),
                          output_format = fmt)

    fs::file_delete("_output.yml")
  }
}
```

And just refer to the two differently formatted ref docx's within `_word_output_numbered.yml` and `_word_output.yml` respectively. Each one gets swapped in when called upon.

Finally, I needed to put in a table of contents and lists of figures/tables at the start of the document. There really doesn't seem to be an easy way to create ToCs in docx from Bookdown, but fortunately it's just a quick click or two in Word to auto-generate them from document headings or existing document styles. [4] But, I wanted some blank pages at the start, with 'Table of Contents' etc as a heading on each one, so at least I had *space* for all these things set aside automatically that I didn't need to think about.

You *can* include some basic latex functions in Rmarkdown, one of which is `\newpage` to start a new page. But, if I tried adding `# Table of Contents` as a header on a page at the start of the document, when I auto generate the ToC in Word this heading itself gets included in the ToC!

I ended up declaring pandoc custom style fences to give this page the style of 'Title' rather than 'heading', so it won't be included in the auto generated ToC. I also used glue and the output detector from above to only include it in docx output:

```
if(grepl("word", fmt)) {
  word_toc <- glue::glue('
\\newpage
::: {{custom-style="Title"}}
Table of Contents
:::
\\newpage
::: {{custom-style="Title"}}
List of Figures
:::
\\newpage
::: {{custom-style="Title"}}
List of Tables
:::
\\newpage
::: {{custom-style="Title"}}
List of Appendices
:::
             ')
} else {
```

```
  word_toc <- NULL
}

word_toc
```

Remember to escape your forward slashes and your curly brackets in glue by doubling up!

Writing environment

Just as an aside, as great as Rstudio is, I wouldn't really recommend it as the best pure writing environment. You can certainly make it a bit nicer with themes and by editing the font used, but you're not going to get away from its original purpose as a coding environment. While fine for the coding parts of my projects, there was also going to be quite a lot of just plain ol' prose writing, and I wanted to find somewhere a bit nicer to do those parts.

Although Rmarkdown's .Rmd files are theoretically plain text files, I found options to be limited when looking for a writing app that can handle them. Initially I tried using VS Code, as while first and foremost a coding environment, it's highly customisable. In particular, a solarized colour theme, an R markdown plugin, and discovering VS Code's zen mode got me quite a long way towards a pleasant writing environment, but it still felt a little janky.

I've ended up settling on IA Writer as my main 'writing' app of choice. It handles .Rmd files without a problem, feels super smooth, and full-screen provides a nice, distraction free writing environment. It's not particularly cheap, but you get what you pay for. As a bonus, I can use Working Copy to sync my git repo for each project to my mobile devices and continue working on them in IA Writer's mobile apps just as if I was on desktop.

If IA Writer is a bit too steep in cost, Typora is a really nice alternative. It's currently free while in beta, so not sure how much longer that's going to be the case, but it also handles .Rmd files really well. I just preferred some of the library/file management options in IA myself.

On mobile, 1Writer or Textastic are also great choices for iOS, and JotterPad on Android is also great for markdown, although I haven't been able to check it works with .Rmd files myself yet.

Reproducibility

Once I start cobbling together workflows and hacks like those I've detailed in this blog post, I invariably start worrying about stability and reproducibility. [5] What happens when I want to start a new project? Or if I lose or break my laptop and I need to start over again on a new device?

This is where working in code is really handy, and having recently tried my hand at R package development comes in *really* handy. See, now I know just enough about how R works with packages to put my custom project functions – like `build_book()` – in an R/ folder within my project, and have the usethis package create me a DESCRIPTION file that lists my project's dependencies.

I can then also drop all my book building assets – like my custom reference docs for docx files, custom latex preambles [6], bibliography files, the works – into that same project, push it to GitHub, and turn that whole thing into a template repository. [7] Now, anytime I need to create a new Bookdown project, I can just clone that repo, and run `devtools::load_all()` to have my custom `build_book` function back in action!

Taking it a step further, I can use the new-ish renv package to create a `renv.lock` file in my template repo that I can then use to *exactly* re-install the package dependencies I might need

for any book project. [8]

As a side bonus, while reassuring myself in creating a stable and reproducible project setup, this also follows a lot of the steps recommended to turn your research into a research compendium. This means that someone *else* can also download the git repositories I create this way, and similarly, use my renv.lock or package dependencies in DESCRIPTION to very quickly get up and running with a similar environment to mine to test out my work for themselves. I could even take it a step further and include a Dockerfile and link to an online Rstudio server using the holepunch package to recreate it exactly!

Wrapping up

This blog post is really the result of a couple weekends of StackOverflow diving and thinking about what I really need for my next big writing project. It can be a bit overwhelming working with something with as many options as Bookdown/Rmarkdown. The sheer amount of configurations and things like the integration with Rstudio can lead you to forget that, underneath it all, is still just code you can pull apart and make do what you want with a little sideways thinking. I found a lot of questions from people trying to make Bookdown do something from within, when really a little meta-coding with a little function that just swaps out different `_bookdown.yml` files as needed would do the trick.