Four ways to streamline your R workflows

Finding ways to reduce manual tasks when programming, like copying and pasting files or code, can save you time and minimise the risk of errors. This blog post guides you through a few small changes to your R workflow to help reduce manual tasks and streamline your programming workflows in R.

December 31, 2023

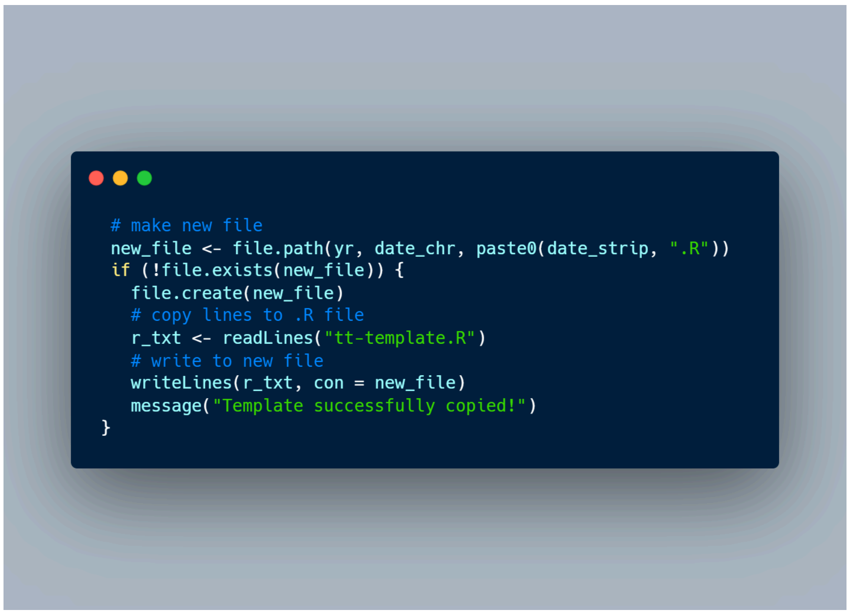
In previous years, when it gets to the end of the year, I've written blog posts about reflecting on a year of #TidyTuesday data visualisations, or making your own #RStats Wrapped in the style of SpotifyWrapped. This year, I wanted to reflect on some of the changes I've made to the way I write R code that have helped to streamline my workflows.

Finding ways to reduce manual tasks when programming, like copying and pasting files or code, can save time and minimise the risk of errors. This blog post will guide you through a few small changes to R workflow that help to reduce manual tasks (basically, a blog post about how to copy and paste less)!

Using template files

Earlier this year, I realised I was doing a lot of copying and pasting of code from one file to another on a regular basis. Like many things in the world of programming, if you find yourself copying and pasting the same thing several times, there is almost certainly a better way of doing it. I noticed that my R scripts for #TidyTuesday always have similar sections: load packages, load data, data wrangling, make a plot, save the plot, and so on - and I was copying and pasting these common sections, then removing the unnecessary parts.

So I decided to created a function that creates these template files for me. The function creates folders based on an input name, creates .R and README files, and pre-populates those files with code. It then also edits that code to load in the correct data based on the function input. I wrote a blog post on creating template files with R if you want to read a little bit more about how I did it.


```r
# make new file
new_file <- file.path(yr, date_chr, paste0(date_strip, ".R"))
if (!file.exists(new_file)) {
  file.create(new_file)
  # copy lines to .R file
  r_txt <- readLines("tt-template.R")
  # write to new file
  writeLines(r_txt, con = new_file)
  message("Template successfully copied!")
}
```

Using GitHub repository templates

Okay, so this isn't technically an *R* specific tip but it has been very useful for saving time in R workflows. Using template files and functions as I described above are really useful if you're creating a small number of files. However, if you're creating multiple directories and multiple files, then GitHub template repositories are really helpful. The idea is similar to template files - starting projects with a skeleton, rather than starting from scratch.

Creating GitHub template repositories is straightforward - you create a repository as normal and then in *Settings*, you can change it to a template repository. Then when you create a new GitHub repository, you can select the option to create from your template repository and the new repository will be pre-populated with the template files. Like normal repositories, GitHub template repositories can be public or private.

I’ve been running workshops reasonably frequently this year, and I typically create a new GitHub repository for every workshop. It contains folders with slides, examples, exercises, solutions, license files, README files, and GitHub actions to deploy slides. Though the content differs for each workshop, the basic structure doesn’t, and a template repository has saved me from having to manually create all of those folders and files each time.

As another example, the Real World Data Science article template created by Finn-Ole Höner was super useful when I was writing my Creating Christmas cards with R blog post.

> Tip: If you’re including GitHub Actions files in the template, then it might useful to disable GitHub actions for the template repository. This means it won’t try to run the actions in the template repository itself.

Linting and styling code with {lintr} and {styler}

Writing code that follows a consistent style can make it easier for people to read, makes collaboration simpler, and can help pick up code errors more quickly. In R, the {lintr} package checks for adherence to a specified coding style and identifies possible syntax errors, then reports them to you so you can take action. I’ve been using {lintr} for a while and it’s definitely helped me to write better quality code - you can also set up GitHub Actions to run linting checks automatically.

This year, I also started using the {styler} package. {styler} goes one step further in terms of code styling, and actually styles your code for you. I used to be a little bit sceptical of packages that overwrite scripts I’ve written, but I’ve quickly grown to love how quick and easy it makes it to style code.

Creating a keyboard shortcut for the `style_active_file()` function has been one of the simplest changes I’ve made. It means I can apply the styling easily (without having to call a function manually or click a specific button) as I’m still developing the code rather than waiting until the end to *tidy up the code*!

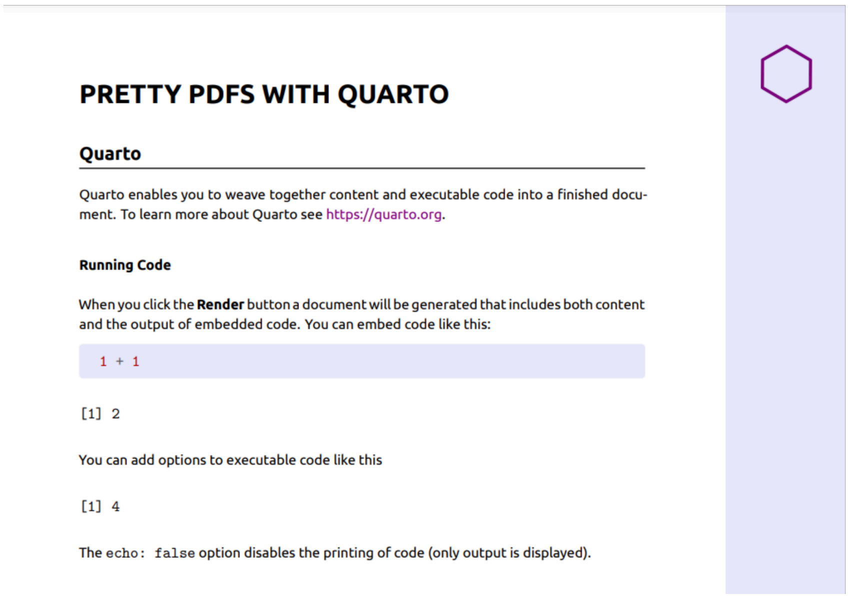
Building Quarto extensions

These days, I make most of my documents and slides using Quarto. If you haven’t used Quarto before, it’s an open-source scientific and technical publishing system that allows you to combine text with code to create fully reproducible documents in a variety of formats - and I highly recommend checking it out. There are lots of ways to customise the way that Quarto outputs look - using built-in options or adding your own CSS styling, for example. Quarto extensions are a way to modify and extend the behaviour of Quarto, including adding additional document styles.

It’s easy to spend too long tinkering with how your documents look (often as a method of procrastinating writing the content…)! So once you’ve settled on a style, having a process to re-use it in multiple documents is an easy way to save time. Initially, I found myself copying and pasting YAML options from document to document, as well as moving .scss and .tex files from one repository to another. Building my own Quarto extensions was a better way of reusing styles across different documents, and also allowed me to keep track of changes to style files more easily.

I wrote a blog post about building a Quarto extension for PDF styling back in February 2023, which details the basic steps for building your own style extension.

I’ll also be running a workshop as part of the *Workshops for Ukraine* series on January 11, 2024, focusing on **Customising slides and documents using Quarto extensions**. You can sign up online to join the session and learn how to build your own Quarto extension.


PRETTY PDFS WITH QUARTO

Quarto

Quarto enables you to weave together content and executable code into a finished document. To learn more about Quarto see https://quarto.org.

Running Code

When you click the **Render** button a document will be generated that includes both content and the output of embedded code. You can embed code like this:

```
1 + 1
```

```
[1] 2
```

You can add options to executable code like this

```
[1] 4
```

The `echo: false` option disables the printing of code (only output is displayed).

What’s next for 2024?

I’m planning to finally start integrating *Nix* into my R workflows, after reading several great blog posts about it by Bruno Rodrigues this year. Nix is essentially a package manager that you can use to install software. With Nix, you can always install the same, specific versions of all the software and R packages you’ve used whenever you build an environment to run your code - making it excellent for reproducible workflows. Part 1 of Bruno’s blog post is a great place to get started!

I started experimenting with Rust towards the end of the year. Rust is a general-purpose, compiled programming language that emphasises performance. By using the {rextendr} package, you can call Rust code from R (much like the {Rcpp} package for calling C++ code from R). I’ve starting re-writing some pieces of especially slow code in Rust, and the performance gains have already been worth the learning curve. To be good at integrating Rust code into R, you need to be good at Rust - so I’ll be continuing to work my way through The Rust Programming Language book in 2024!

How will you streamline your programming workflows in 2024?

Image: giphy.com

For attribution, please cite this work as:

> **Four ways to streamline your R workflows.**
> Nicola Rennie. December 31, 2023.

nrennie.rbind.io/blog/four-ways-streamline-r-workflows

2 Comments - *powered by utteranc.es*

mwangi-george commented on 31. Dez. 2023

I have always wanted an easier way to use the styler package. Thanks for keyboard shortcut tip 👏

👍 1

njtierney commented on 8. Jan. 2024

Great post! I have now changed my keyboard shortcut from "style selection" to "style active file" .

🎉 1

Write Preview

Sign in to comment

Styling with Markdown is supported Sign in with GitHub

← Making art in Python with plotnine Making Pretty PDFs with Typst (and Quarto) →