Department
for Education

DfE Statistics Development Team Workshops

Coding RAP using R

Contents

Introduction

We’ve prepared this walkthrough guide for statistics publication teams as an introduction to the ways in which coding in R can be used for Reproducible Analystical Pipelines (RAPs), creating functions for typical tasks that teams may come across. The guide is intended to be step-by-step, building up from the very basics. The plan is to work through this in groups of 3-ish with access to experienced R users for support. If it starts too basic for your level, then just go through at your own/your group’s pace as you see fit. By no means can we cover everything in this walkthrough, so please see it as a prompt to ask follow-up questions as you’re working through on anything related to R, RAP and coding in general.

What is a RAP?

RAP stands for Reproducible Analytical Pipeline. The full words still hide the true meaning behind buzzwords and jargon though. What it actually means is using automation to our advantage when analysing data, and this is as simple as writing code such as an R script that we can click a button to execute and do the job for us.

Using R (the coding language) really helps us to put the R in RAP (‘reproducible’). Ask yourself, if someone else picked up your work, could they easily reproduce your exact outputs? And when the time comes around to update your analysis with new data, how easy is it for you to reproduce the analysis you need? In an ideal RAP, it would be as simple as plugging the new data in and clicking ‘go’, with no need to manually scroll through multiple scripts updating the year in every file name or the variable name that’s changed from using _ to -.

What is R/R Studio?

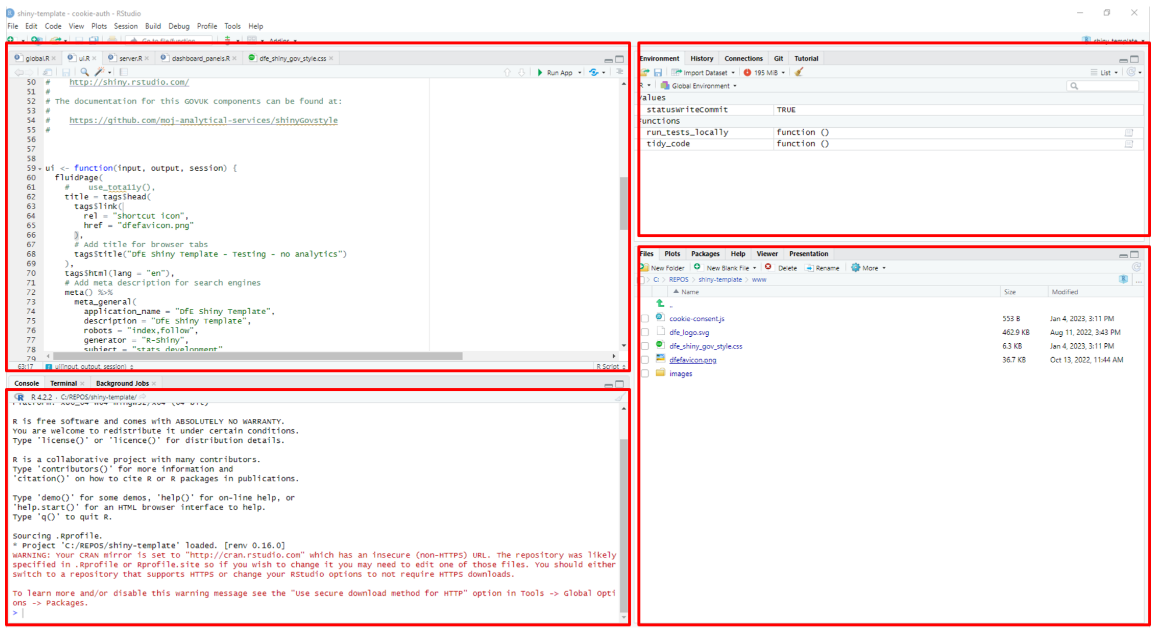
R is a coding language. There are many different languages of code, some others include SQL, python, JavaScript and many more. They all have benefits, and the difference is often the syntax used (literally like learning new languages!). R is open-source, meaning it is free, anyone can use it and anyone can contribute to developing new ‘packages’.

- A *package* is a set of functions someone else has written and tied together in a nice neat bow, ready for you to use! You simply install the package, and then you have all of the functions available.

- A *function* is a chunk of code that has been grouped together, given a name, and often has 'place holders' you can change, such that you can use that name to run that code, and apply it to different data. (for example, mean(x) is a function that calculates the arithmetic mean of x. You replace x with any numeric data.)

R studio is a programme that enables us to code in the R coding language, while also providing a visual and interactive interface. It has useful areas and windows that enable you to see what tables you have loaded, charts you have created, Git user interface, file explorer, help windows and many more! Normally you will see the screen split into 3 or 4 windows:

Figure 1: R studio panels



```
#    http://shiny.rstudio.com/
#
# The documentation for this GOVUK components can be found at:
#
#    https://github.com/moj-analytical-services/shinyGovstyle
#



ui <- function(input, output, session) {
  fluidPage(
    #    use_tota11y(),
    title = tags$head(
      tags$link(
        rel = "shortcut icon",
        href = "dfefavicon.png"
      ),
      # Add title for browser tabs
      tags$title("DfE Shiny Template - Testing - no analytics")
    ),
    tags$html(lang = "en"),
    # Add meta description for search engines
    meta() %>%
      meta_general(
        application_name = "DfE Shiny Template",
        description = "DfE Shiny Template",
        robots = "index,follow",
        generator = "R-Shiny",
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Sourcing .Rprofile.
* Project 'C:/REPOS/shiny-template' loaded. [renv 0.16.0]
WARNING: Your CRAN mirror is set to "http://cran.rstudio.com" which has an insecure (non-HTTPS) URL. The repository was likely
specified in .Rprofile or Rprofile.site so if you wish to change it you may need to edit one of those files. You should either
switch to a repository that supports HTTPS or change your RStudio options to not require HTTPS downloads.

To learn more and/or disable this warning message see the "Use secure download method for HTTP" option in Tools -> Global Opti
ons -> Packages.
> 
```

```
Values
  statusWriteCommit    TRUE
Functions
  run_tests_locally    function ()
  tidy_code            function ()
```

Name	Size	Modified
..		
cookie-consent.js	553 B	Jan 4, 2023, 3:11 PM
dfe_logo.svg	462.9 KB	Aug 11, 2022, 3:43 PM
dfe_shiny_gov_style.css	6.3 KB	Jan 4, 2023, 3:11 PM
dfefavicon.png	36.7 KB	Oct 13, 2022, 11:44 AM
images		

- Top left - Source pane. Open and view your code scripts, tables, data sets, functions.

- top right - Environment/History/Connections/Git (if you have it). The environment shows you what data, functions, objects etc you have.

- Bottom left - Console. Shows what you have run, and you can type and run commands directly into the console.

- Bottom right - File explorer/plots/packages/Viewer.

Pre-workshop requirements

Technical requirements

First of all, make sure to bring your laptop. This is going to be interactive and require you to do some coding.

Preferably before coming along, you'll need to go through the following list of things you'll need to make sure are set up on your DfE laptop:

- Set up an Azure Dev Ops Basic account (not a Stakeholder account) at the DfE Service Portal; Either:
- Install git on your laptop: https://git-scm.com/downloads;
- Install R-Studio on your machine: Download **R for Windows (x64)** and **RStudio** from the Software Centre on your DfE laptop.

Or:

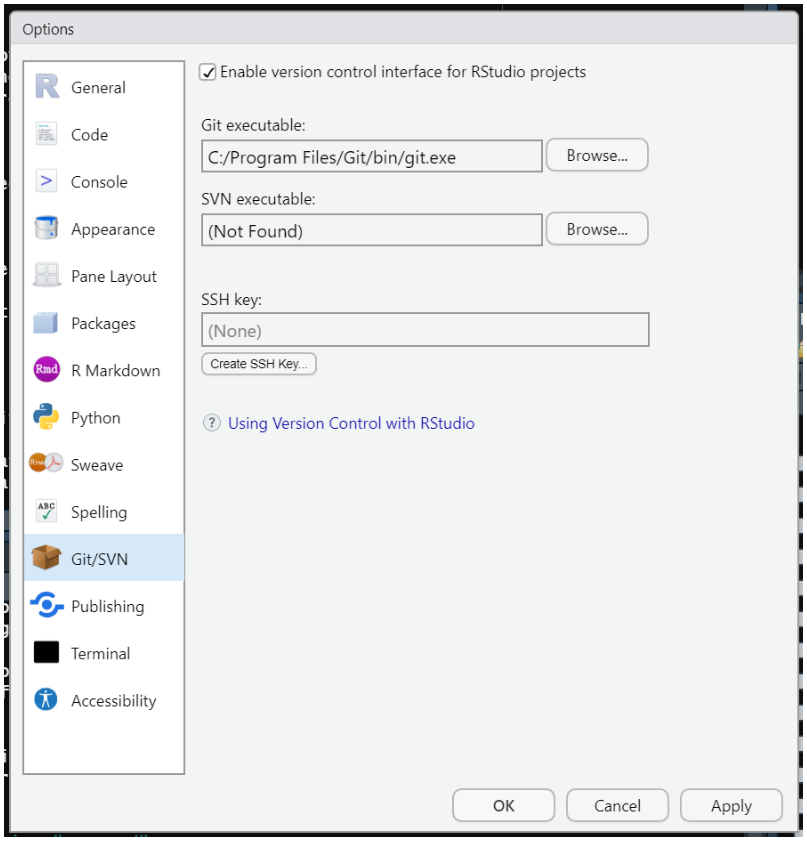
- If you're on EDAP and used to using R/R-Studio and/or git on there, feel free to just use that.

You'll also need to make sure that git is set up in the git/SVN pane of global options in R-Studio (found in the Tools drop down menu). Make sure the path to your git executable is entered in the git path box and git should automatically be integrated with R-Studio.

Once you open a repository, you'll get an extra panel, named 'git', in the top right pane of R-Studio and you'll also be able to use git in the 'Terminal' tab at the bottom left (in the same place as the R console).

A useful thing here if you want to use git commands in the terminal is to switch the terminal from the default Windows Command Prompt to `git BASH`. You can do this in the Terminal tab of R-Studio's global options - just select `git BASH` from the 'New terminal opens with' pull down menu. Click apply and then select the Terminal tab (next to the Console tab), click 'Terminal 1' and then select 'New terminal' from the drop down menu. You should see something similar to the terminal screenshot.

Figure 2: Enter the path to your git executable in the git path option box



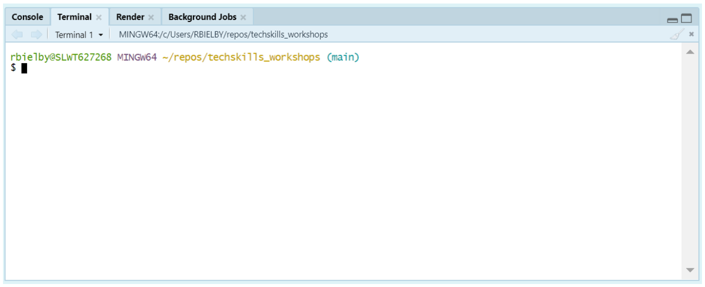
☑ Enable version control interface for RStudio projects

Git executable: C:/Program Files/Git/bin/git.exe

SVN executable: (Not Found)

SSH key: (None)

Figure 3: The ‘git BASH‘ terminal in R-Studio



```
MINGW64:/c/Users/RBIELBY/repos/techskills_workshops

rbielby@SLWT627268 MINGW64 ~/repos/techskills_workshops (main)
$ 
```

Working in teams

To get the most out of git and Dev Ops, you're going to need to work in teams. We're aiming for groups of 3. Some of the tasks we'll work through will require just one of your team to perform, whilst others will require all of your team to perform them. If it's not clear then ask and most importantly, communicate with each other about what you're doing.

To help illustrate the challenges and benefits of git and Dev Ops and how to work collectively within the same space, all the groups will be working within the same repository. Each group will have their own working branch (already created) with the naming format `workshop_group_N`:

- `workshop_group_1`
- `workshop_group_2`
- ...

We'll need to prefix branches and tasks within the repository with `grN_` and group N respectively (again switching the `N` for your group number).

By the end, you should get a good idea of how to utilize R for RAP processes, as well as how to collaborate using git!

Getting started . . .

Now we will begin the tasks and group work. First, everyone **open up R studio.**

Creating a project

When creating RAP processes, it's key that your code is reusable, transparent, and well documented, such that it is future-proofed for any future team members. Therefore, first, we will show you how to create an R project. Using an R project ensures that all of the files in the project folder (scripts, plots, notes, etc.) can all be referenced relative to the **.Rproj file** location. Basically, it removes the need to set and get your working directory, and removes the need to use long & complete file paths. Instead, you essentially start any file path from the location of the .Rproj file! *For example*, if you have an code.R code script and a data.csv file in a project folder called 'workshop' that has been set up as an R project, if you want to read the .csv into the R script, you don't need the entire file path of the csv, just the file path relative to the .Rproj fil, which in this case would simply be 'data.csv'.n

If you're struggling to grasp that, that's okay, we will create a project and see this in action now. To create a new project, in R studio, go to **File > New Project > New directory > New Project**. Give your project a name and choose where about in your files you would like the project to be saved. Click 'Create project'.

Using renv

Renv (short for R environment) is a package in R that helps you to keep a record of which packages your project uses, and what version of each package it should use. Using renv in your R project means that anybody in the future (including your future self) who comes back to this project, can immediately get all of the required packages. First you need to install the renv package by running `install.packages('renv')` in the console.

Once that has installed, you can run the functions from this package. You can also see what packages are installed by navigating to the 'Packages' tab in the bottom right window - renv should have appeared in here now.

When using functions from packages, you can either type `the package name::the function name` or just the function name on its own. However, problems can arise if two packages have a function with the same name, so using the package name and colons is safer.

To activate renv, run `renv::activate()` in the console. If you navigate back to the files tab in the bottom left and look in your project folder, you should see some renv-related files have appeared, including a *renv.lock* file. The renv.lock file is the file that records all of your project's packages and their versions. If you click on the renv.lock file it will open for you to view in the top left.

Each time you add or remove packages, or update the package versions, you should run `renv::snapshot()` to take a snapshot of the current state of your library. If you ever get a new device, or someone else wants to view your project, they simply need to run `renv::restore()` to restore the package library on their device to match the packages needed in the project.

Your initial script

Now that we have your project created and renv activated, we can create our first code script. Open a new R script file by selecting **File > New File > R script**.

Comments and headings

Comments in code are some of the most useful documentation you can use, and are crucial for RAP! You should start your script with comments that provide important information such as what this code script contains and what it is for - basically anything you think a new person would need to know in order to understand and run the code effectively. You add comments in R scripts by starting the line with a **hash symbol, #**. Any line that starts with a hash symbol will be treated as a comment rather than code, and so will not be 'run'. You can also add comments to the end of lines of code by including the hash symbol half-way through a line - everything after the # will be treated as a comment, and everything before will be treated as code. If you have multiple lines of comments you wish to add, you can type them out, highlight them and use `ctrl+shift+C` to 'comment out' all of the highlighted lines.

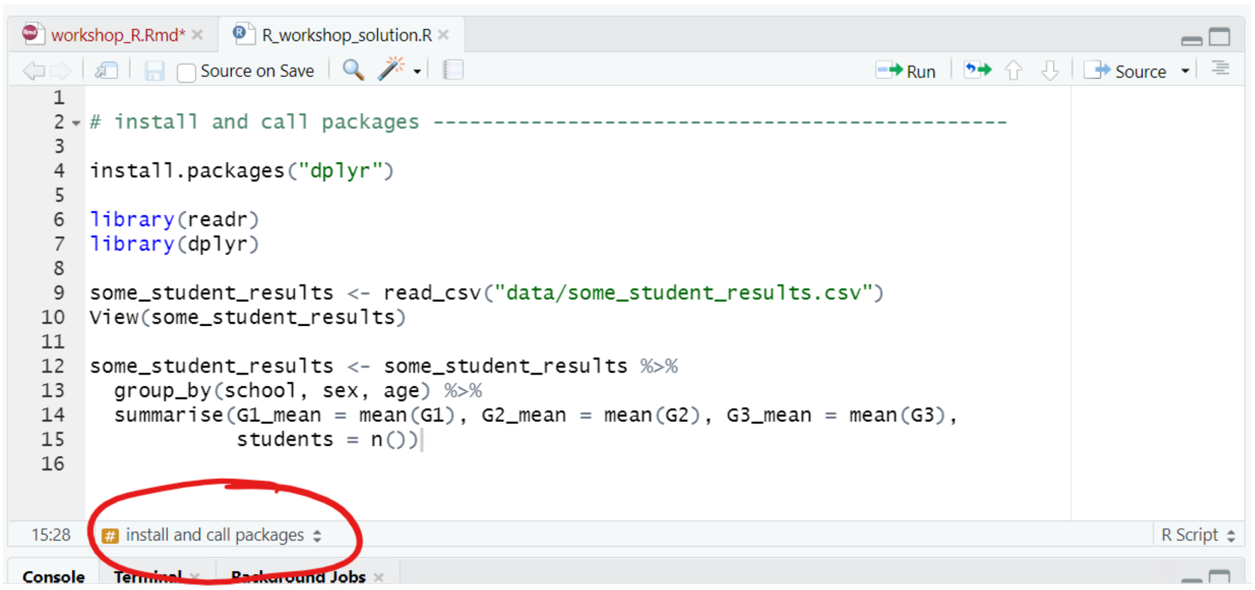
Add some comments to the beginning of your script now - include your name, the date, and a short description that explains that this is the first script, and will include library calls as well as loading data and any functions that will be used in future.

Your first script should definitely include the 'library-calls' for all of the packages you will need. This basically means loading any packages you need at the beginning, before running any future code that requires them. We load/call packages by using the `library()` function.

Our first steps only require two packages, 'readr' and 'dplyr'. You can either install these by typing `install.packages('readr')` and `install.packages('dplyr')` one at a time in the console, or you can install both packages at once by running `install.packages(c('readr', 'dplyr'))`. Then, remember to take a `renv::snapshot()`!

Now, below your initial comments, we will add the library calls to the script. First, we should add a section header! Section headers are yet again another form of good documentation, and and extremely helpful when navigating around scripts of code! To add a section header, we can use the keyboard shortcut `ctrl+shift+R`. In the pop-up window, give the first section the header 'Load in packages'. Then click okay and you should see the header has appeared in your code. In the bottom left of the code window, you should see the section header has appeared, and if you click on the name you can navigate to other sections when you have built up your script further.

Figure 4: R studio panels



```
# install and call packages -----------------------------------------------

install.packages("dplyr")

library(readr)
library(dplyr)

some_student_results <- read_csv("data/some_student_results.csv")
View(some_student_results)

some_student_results <- some_student_results %>%
  group_by(school, sex, age) %>%
  summarise(G1_mean = mean(G1), G2_mean = mean(G2), G3_mean = mean(G3),
            students = n())
```

Another way to view and navigate between all of the sections in a script is to use the keyboard shortcut `ctrl+shift+O` to open the sections in a small panel on the right of your script.

Adding and running code

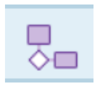
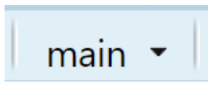
Now, below the heading on a new line, add `library(readr)`, then on a new line add `library(dplyr)`. In R, each new line represents a new code command. You don't need to use a semi-colon or any other punctuation to separate commands other than just starting a new line, and you cannot have two commands on the same line. Therefore, the two library calls are two distinct commands, as they load different packages, and so need to each be on a new line.

Summary

We've looked through a lot of the basics in this section, covering adding/staging, committing, pushing/pulling between remote and local repos, merging and pull requests. These are all the main concepts you need to use git.

We've also tried to cover doing all this through a mixture of RStudio, git BASH and GitHub (and as we've said Azure Dev Ops offers similar functionality to GitHub). Most common processes can be done multiple ways and there's not necessarily a single right method to follow, just whichever makes most sense in your situation.

Just a quick final note on why it's useful to be familiar with git `BASH`. Whilst most of the basic git functionality can be accessed via the RStudio panel or GitHub/Dev Ops, there are some things that are best achieved through BASH. In particular, if you have a file in your repo that you need to remove entirely, this pretty much requires someone to use commands via git BASH.

Process	git BASH	RStudio git panel
Create branch	`git checkout -b branch_name`	
Switch branch	`git checkout branch_name`	
Merge branch	`git merge branch_name`	N/A - use GitHub/Dev Ops

Troubleshooting

renv

If `renv::restore()` causes issues, then one of your team should try `renv::init()` and select option 2 to restart renv. Then do a add/commit/push cycle and get the other team members to do a pull and then try running `renv::restore()` again on their local clones of the repo.

Datafiles commit-hooks/`.gitignore`

To help teams keep on top of avoiding any accidental publishing of unpublished data, we've added in some code around commits that checks through any data files in the repo and checks them against a logfile and the .gitignore file. Any files listed in .gitignore will not be included in commits and therefore won't be sent to the remote repo as part of any push.

merge conflicts

Merge commits happen when two branches have conflicting changes that have been made concurrently. `git` can usually figure out how to prioritise changes based on the commit history, but if changes have happened at the same time to the same bit of code across different branches, then it will need to get your input on how to prioritise the changes.

The easiest way to go through how to deal with merge conflicts is by discussing with an example, so ask us in the workshop if and when you hit a merge conflict.

Briefly though, when there's a merge conflict, git will add some text to the file containing the conflict along the following lines:

```
<<<<<<<<<<< branch_1
code
on
branch
1
```

```
===========
conflicting code on branch 2
>>>>>>>>>>> branch_2
```

Effectively as the user, you need to decide which bit of code is the right bit to keep and then delete anything you don't want to keep as well as the tag-lines that git has added in. So for example, you should be left with something along the lines of:

```
code
on
branch
1
```

Once you've cleared up all merge conflicts in the branch that you're working on, then perform another add/commit cycle and thay should clear out the conflict from the branch that you're working on and you'll be able to continue with the intended merge/PR.

Department
for Education