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 Analytical 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 -.

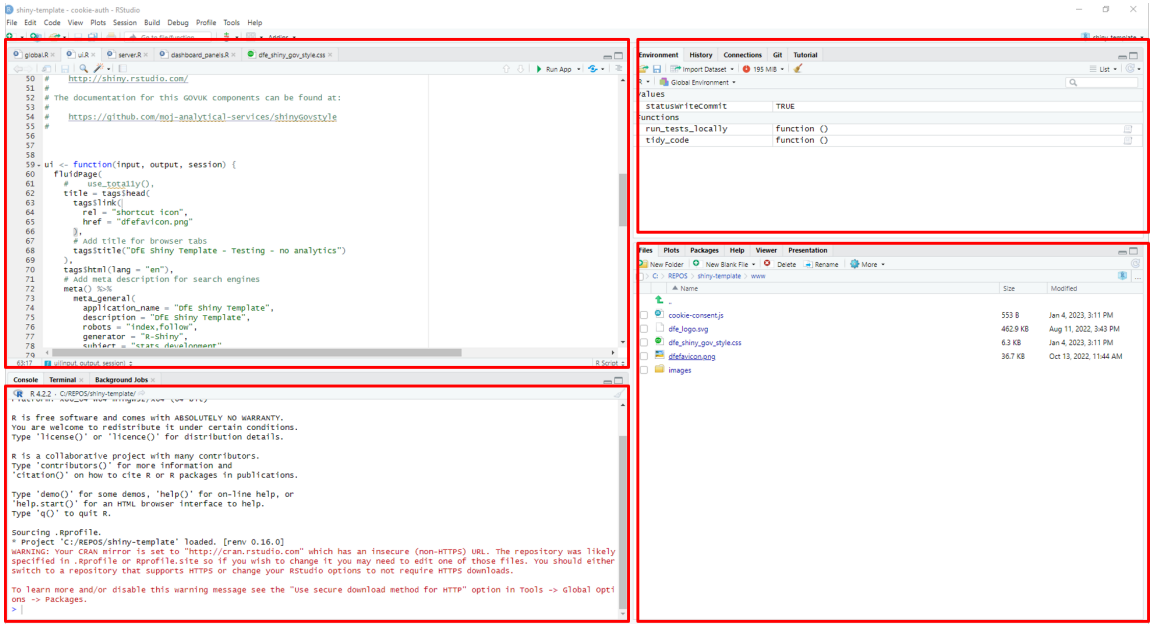
What is R/R Studio?

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



- 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

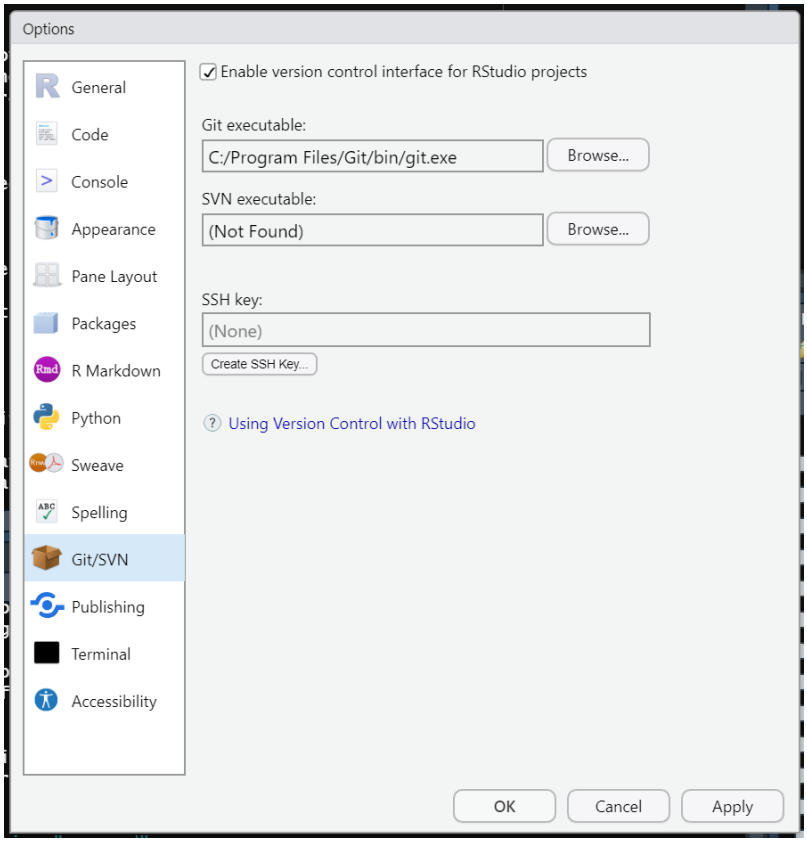
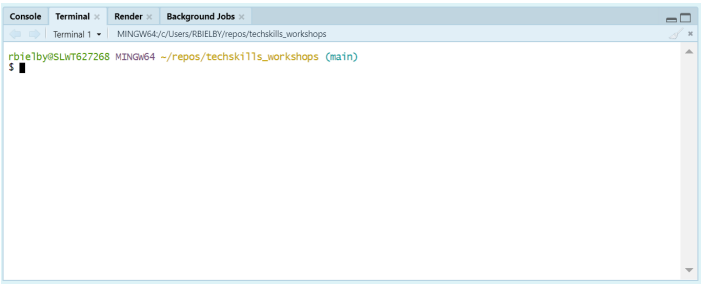


Figure 3: The ‘git BASH‘ terminal in R-Studio



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 a code.R code script, and a data.csv file in a project folder that has been set up as an R project, and you want to read the .csv in the R script, you don't need the entire file path of the csv (which probably starts C:/. . . and can get very long), just the file path relative to the *.Rproj file* which in this case would simply be 'data.csv'.

If you're struggling to grasp that, that's okay, we will create a project and see this in action today. 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'. We aren't going to discuss using Git or version control today, but if you are using these in a project, make sure you **do not** save the project folder in OneDrive (this is because OneDrive has it's own version-control system, and trying to use Git within that can cause complications and errors).

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.

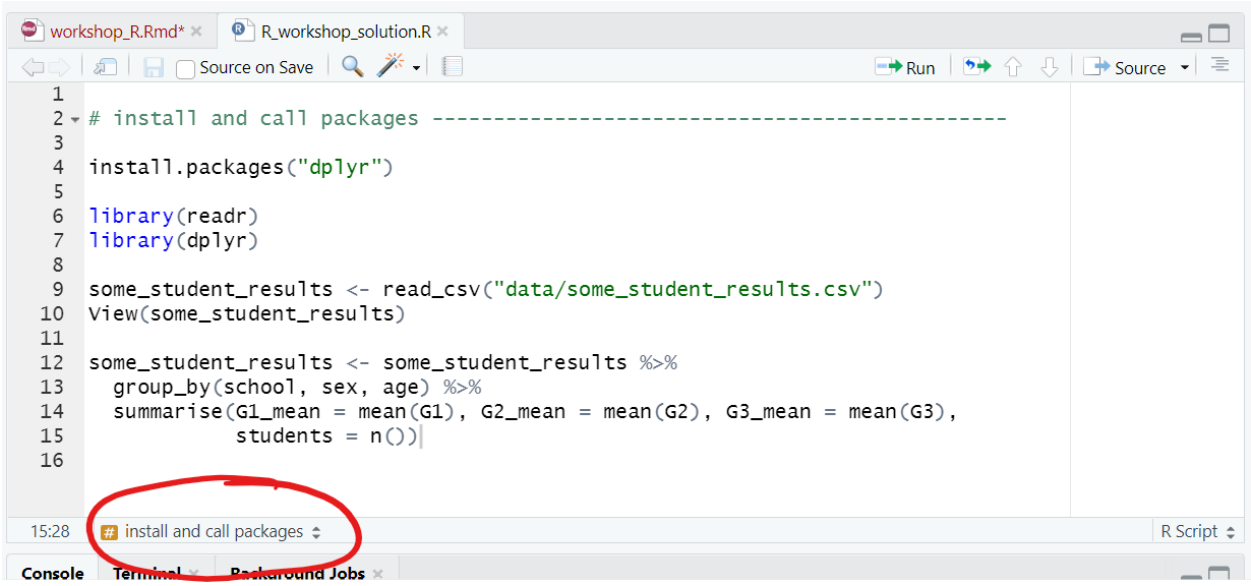
Add some comments to the beginning of your script now - include your name, the date, and a short description that explains that this is the first script, and will include library calls as well as loading data and any functions that will be used in future.

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 section headers navigation



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.

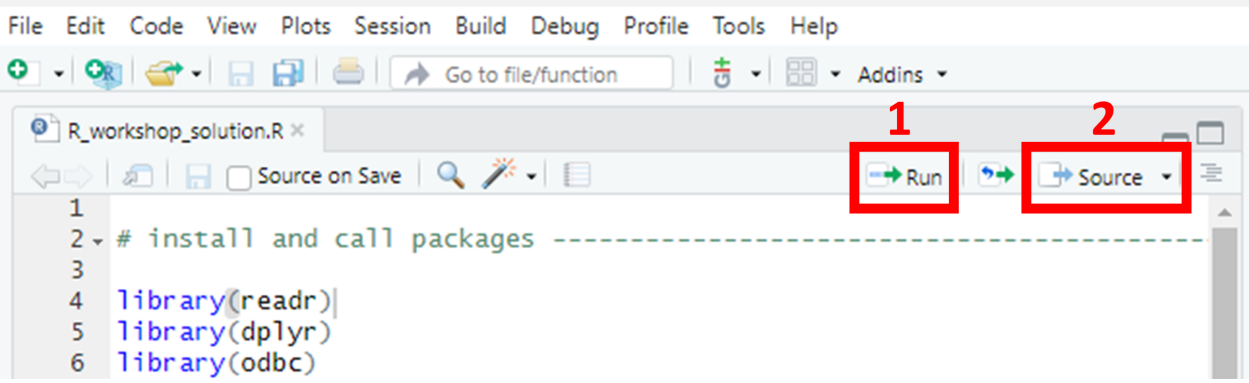
Throughout this workshop, make sure to add comments at every step so you know what each code snippet is doing, and why!

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, since the two library calls are two distinct commands because they load different packages, they each need to be on a new line.

To actually run code that you have saved in a script file, there are a few options. Firstly, there are buttons you can use to the top right of the script file's window.

Figure 5: R Studio run and source buttons



Button 1 in the above image (the 'run' button) will run either all *highlighted* code, or, if no lines are highlighted it will run the line or snippet where your cursor is located (in the above image, you can see the faint cursor line at the end of line 4, so clicking run would run line 4 in this instance). Another way to do this is to use the 'run' keyboard shortcut - `ctrl + Enter`. This works in the same way and will either run all highlighted lines, or the snippet where the cursor is located.

Button 2 (the 'Source' button) will run the entire script from the first to the last line in one go! This is the same as running `Source("script_name.R")` in the console. `Source(...)`

is particularly useful in RAPs, as one of the baseline expectations for statistics publication pipelines is that you have one script that runs every single step of the process - sourcing scripts one by one in the correct order in a 'run_pipeline' script (with comments!) is great way to do this while keeping it tidy and clear!

Highlight the lines you have just added to the script and run them either using the button or the keyboard shortcut. Now we have the packages we need to load in some data.

Loading in the data

There are multiple ways to load data into R. Today we will cover two common ways we see analysts use;

1. Reading in a **CSV** file,
2. Loading data in directly from **SQL** servers.

Before using either method to load in the data, we should add a new section header! Use the guidance above to add a new section header called 'Load in the data'.

Reading in CSVs

We already installed the `readr` package, which contains the functions we need to read CSVs into R. In R, i is best practice to name our objects (tables, data sets, etc) so that you can use that name to reference the objects you need. This means that while the code `readr::read_csv(insert_filename/filepath_here)` is the function that reads in a given csv, we should write it in the below format:

```
chosen_name <- readr::read_csv(insert_filename/filepath_here)
```

Then, if you navigate to the 'environment' window in the top right of the screen, you should see an object has appeared with your chosen name under the heading 'Data'. If you click the blue circle with the white arrow inside, it will expand to show you information about the data you've uploaded, including the name of every column/variable, and the format of each column (i.e. chr for character, num for numeric). If you click the name of the object, it will open in the viewer in the top left so that you can look at the table. This is the same as typing `View(name_of_data)` in the console.

Reading in from SQL

In order to read the data in from SQL we need to install the relevant packagesand connect to the SQL database. Firstly, we need to install the `odbc` and `DBI` packages. Try to use the above guidance to;

a. Install the `odbc` and `DBI` packages,

b. Add the library calls to the correct section of the initial script,

c. Take a snapshot to update the renv.lock file.

Once you have complete the above steps, we can use the new packages to connect to the secure SQL database. We use the following code (inserting the server and database we require) to create an object with the connection. You can call it anything, but common practice is to call this connection 'con':

```
# Connect to SQL database
con <- DBI::dbConnect(odbc::odbc(),
    Driver = "{SQL Server Native Client 17.0}",
    Server = "...",
    Database = "...",
    Trusted_connection = "Yes"
  )
```

[Note that this code will only work for you if you have access to the specified server and database, so only users with existing access can use this code in R to connect and pull in data! If someone without access tried to run this code, they would get an error in the R console.]

Now that we have the connection stored and named, we can use it to read in a table from the database into R. There are multiple ways to do this - in this workshop we will cover two of them.

1. Use `tbl(con, table_name)`,

2. Use `sqlQuery(con, "SELECT * FROM [table_name];")`.

Option one above is a very concise and tidy way, which (as long as you're add descriptive comments) will make your code easy to read and understand. Option two is slightly longer, however, it allows you to run SQL queries in R, so if you have previous experience with SQL you may find this useful. You can edit and customise the SQL query, meaning you have the option of selecting specific columns and filtering at this stage, rather than reading in the entire table. Just ensure that you write the table_name exactly how you would in SQL (including schemas,)

Whichever option you choose, you need to choose what name you would like this object to have in R once the table is read in. It is best practice to make object names descriptive but concise. This means describing what the table contains, but not including unnecessary and repetitive words such as 'table' in the name. The data we are using today contains data on the characteristics and grades of some students, so we might choose to call the table 'students'. We assign the table this name by using the <- symbols, so the full snippet should look something like this:

```
# Read in student data from SQL
student_results <- tbl(con, table_name)
```

Add this under an appropriate heading/comment to your initial script. Then, if you navigate to the 'environment' window in the top right of the screen, you should see an object has appeared with your chosen name under the heading 'Data'. If you click the blue circle with the white arrow inside, it will expand to show you information about the data you've uploaded, including the name of every column/variable, and the format of each column (i.e. chr for character, num for numeric). If you click the name of the object, it will open in the viewer in the top left so that you can look at the table. This is the same as typing `View(name_of_data)` in the console.

Manupilating data

Once you have pulled data into R studio, it is likely you want to manipulate it in some way. In this workshop, we have pulled in data that contains one line per student, so one common task is to aggregate data, and get it in the correct tidy format for EES.

Aggregate & filter data

Aggregating data is a common task analysts in in statistics publication face. In the data we have loaded into R studio in this workshop, there is currently one line per student, containing that students characteristics and grades. We want to aggregate this by certain characteristics and get the average grade for each aggregated group. Firstly, it's possible that we might want to filter on certain variables so that we only include certain students in our data.

To **filter** data, we can simply use the `filter()` function from the `dplyr` package, which we have already installed and loaded. Here we will also introduce the **pipe** symbol - %>%.

In our environment, we should see the data we have loaded in with the name we gave it - 'student_results'. If we want to filter this, we would do so in the following format:

```
student_results %>%
filter(column_name == contraint,
      column_name2 < constraint2,
      column_name3 %in% c(list1, list2, list3))
```

You can give this a go in the console without storing the results in the environment by simply running the above without giving it a name like we did when we read the data in. If you wanted to store your results, you would add `name <-` at the start of the first line. In today's workshop, we don't need to filter the data before we start aggregating. The first thing we want to do is specify the variables we want to group the data by. In this example we want to group by **year, school, sex** and **age**. This is very easily done using the `group_by()` function as below:

```
student_results %>%
group_by(year, school, sex, age)
```

However, the output from running the above code will look like it hasn't changed, because it doesn't know how to aggregate the groups yet. We define this using the `summarise()` function *after* the `group_by()` function. In this workshop we will use two common aggregations; averages and counts, however, this method can be applied to all sorts of aggregation methods!

Today, we want to aggregate the data such that we have a row for every group of year, school, sex and age, with the summary statistics being the mean grades (G1, G2, G3)

and the total count of students in each row. To start with, we will lay out the code in an explicit form, before we discuss how to convert it to best practice. Running the following code in the console will give the output we want:

```
student_results %>%
  group_by(year, school, sex, age) %>%
  summarise(G1_mean = mean(G1),
            G2_mean = mean(G2),
            G3_mean = mean(G1),
            students = n())
```

Within the `summarise` statement, it's defining 4 new columns. `G1_mean = mean(G1)` means that the new column called G1_mean will be the mean of the previous G1 column when grouped by the variables defined in the `group_by()` statement. `students = n()` creates a new column named students, which contains the number of rows that have been included in each group.

While the above code works, it can be written in many different ways, and while in this example we are only creating 4 summary columns and using 4 grouping variables, in real life this can get messy the more columns and variables we add! Copying and pasting functions and lines and changing the names each time is prone to mistakes. For example, did any of you notice that **there is a mistake in the above code**? I forgot to update the final mean, which should have been taking the mean of the G3 column! However, in my haste of copying and pasting the same thing 3 times, I only updated the name to G3_mean, while forgetting I also needed to edit what was in the mean() brackets.

In real life, mistakes like this are more common than you'd think! There are best practices in writing our code that can help us to avoid mistakes like this. If you're applying the same function to multiple columns like we are with `mean()` above, we can use `across()`. The `across()` function in R allows you to apply one function across a list of columns, and also enables you to specify the new column's names! We can therefore rewrite the above code as below:

```
student_results %>%
  group_by(year, school, sex, age) %>%
  summarise(across(c(G1, G2, G3), mean, .names = "{.col}_mean"),
            students = n())
```

In the above, we have *nested* an `across()` function inside a `summarise()` function. `across(c(G1, G2, G3), mean, .names = "{.col}_mean")` means apply the `mean()` function to G1, G2 and G3 one-by-one, and set the new column name to be the original column name followed by `_mean`. In this instance it might not seem much simpler than the first version, however using 'across' can be extremely useful when real-life examples have extremely large data sets! Lets store the above results in our environment now and call them `student_results_aggregated` using the following code:

```
student_results_aggregated <- student_results %>%
  group_by(year, school, sex, age) %>%
  summarise(across(c(G1, G2, G3), mean, .names = "{.col}_mean"),
            students = n())
```

Reorder and rename columns

Something that some analysts might leave to the end or do manually in excel is reorder and rename columns. However, even simple steps such as are prone to human error and can result in big mistakes! It also means that your process isn't fully **automated** or **reproducible**, since someone new to your project wouldn't know from looking at it what you have done, and therefore wouldn't be able to recreate it! To avoid this, it's always best to do these simple steps in the code. It also makes your own code robust - if your code for analysis expects columns to have certain names, to all be lower case, or to be in a certain order, and they aren't, it will cause errors later down the line!

First, a very useful function to use is `variable.names()`. Try running `variable.names(student_results)` and `variable.names(student_results_aggregated)` in the console. It should return a list of all of the variable names in the specified tables. You can also use this to compare the names of two data sets by running `variable.names(student_results) == variable.names(student_results_aggregated)`. This will return a `TRUE/FALSE` list - you will see in this case, the first 3 are `TRUE` - this is because the names of the first 3 columns are school, sex, age in both tables, however the rest differ. This is useful when you are *expecting* two tables to have the same variable names, for example in an annual publication cycle, it is a good idea to check that the variable names this year are the same as they were last year before you start your analysis.

First, lets reorder the columns. I want the 'students column to appear *before* the grade

means. We can cimply use the `select()` function here, listing the columns in the order we wish them to appear:

```
student_results_aggregated %>%
  select(year, school, sex, age, students, G1_mean, G2_mean, G3_mean)
```

Now if we also wanted to rename columns, we can do it together in one step by using another 'pipe' (%>%). If there are only a select number of columns you wish to rename, you would use `rename` and specify as follows:

```
student_results_aggregated %>%
  select(year, school, sex, age, students, G1_mean, G2_mean, G3_mean) %>%
  rename(g1_mean = G1_mean, g2_mean = G2_mean, g3_mean = G3_mean)
```

So when using `rename` as above, the order needed is `desired_name = original_name`. As you can see above, all this code does is convert the upper case G's to lower case, however, you could rename each column to anything you like using this method. It's best practice to stay consistent with the case you use, and in EES, we expect all column names in 'snake_case'. Therefore, something you might want to do is rename all of the columns to lower case. If your rename-step is something like converting all columns to lower case, this can be done in a more effective way, with less copying and pasting, using `rename_all()` and `tolower()` like this:

```
student_results_aggregated <- student_results_aggregated %>%
  select(year, school, sex, age, students, G1_mean, G2_mean, G3_mean) %>%
  rename_all(tolower)
```

Since above, we have started the first line with `student_results_aggregated <-`, this means it's going to overwrite whatever was stored in the environment under that name before! This *can* be risky, since `select()` can also be used to **drop** columns. If you left out any names from the select line by accident, they would simply be dropped, so be careful if you use the same name as the object from before this step! This can be avoided by picking a different name, like `student_results_aggregated_reordered` for example. If you *did* make a mistake, and overwrote something by accident, you can just re-run the code that made the original object before it was overwritten.

Suppression (and writing functions)

Another common task that analysts face is suppressing data. In this section, we will write our own function that will do the suppression step for us! We have already been using functions from *packages* throughout this workshop - everything in the format `function_name()` is a function that has already been written for us - it takes whatever we put in the brackets, does things to it, and creates an output. Here we will introduce writing our own!

Writing functions

The best way to introduce writing functions is by starting very simple to understand the concept first, so let's practice here with some easy examples. Open a **new R script** (file > New file > R script) and call it 'functions.R'. On the first line, add some **comments** that explain this script is to practice writing functions.

Firstly, we will write a simple function that adds two numbers together. Give it a heading that describes this. On a new line, we will use the following syntax to create our first function:

```
function_name <- function(inputs...){
function instructions...
}
```

In our function, we want to add two numbers, so we want two inputs. We can call them whatever we want, but you should choose something logical that makes it easy to understand. Whatever names you choose for the inputs in the `function(...)` bracket, you then use those names as place-fillers in the instructions. Then, whenever the function is used, it will carry the inputs through the instructions in the right places. Here is an example:

```
add_together <- function(x,y){
x + y
}
```

The above function takes two inputs, x and y, and adds them together. To *use* the function, you need to run the above code once to *create* it, then you can use it in the

console. You should also see it appear under 'functions' in your environment! Once it's creates, you can use it in the console by running `add_together(2,3)` as an example, giving the output of 5 if it's been written correctly! You can choose whatever numbers you like to replace x and y when using it in the console as above.

Functions can also include character inputs as well as numeric! Another example we will use here to practice is creating a function that outputs a sentence. a simple way to output text in R is to use the `paste()` function. Try creating a function called 'introduction_text', that has *two inputs* defined as 'name' and 'age', and uses `paste("Hi, my name is", name, "and my age is", age)` as the *function instruction*. Test it out! Do you see your sentence appearing in the console?

if() and ifelse() statements

`if()` and `ifelse()` statements work similarly to using IF in excel formulas - they apply tests and perform actions based on the outcome of the test. In the console, if you type a question mark followed by a function name it will open the help page for that package! Try it out here by typing `?ifelse` and read the help page to understand the syntax.

The difference between `if()` and `ifelse()` is that `ifelse()` statements perform one yes/no test - for example `ifelse(age >= 16, '16+','under 16')` tests whether age is greater than or equal to 16. If it is, then it outputs '16+'. If it is not, then it outputs 'under 16'. If statements can have multiple conditions and can be stacked. For example;

```
if(age >= 16){
'16+'
} else if(age < 12){
'under 12'
} else if(12 <= age < 16){
'12 - 16'
} else NA
```

Here there are 3 'tests' and 4 different outputs. The outputs are either '16+', '12-16', 'under 12' or NA. You can also have multiple conditions in one `if()` statement using *and* (`&` symbol) or *or* (`|` symbol) depending on what you need. For example;

```
if(age >= 16 & sex = 'Female){
```

```
'16+ female'
}
```

This test is finding instances where age is greater than or equal to 16 AND sex is female. We might want to use *or* in cases like the following example:

```
if(object == 'Bread | object == 'Fish'){
object == 'Food'
}
```

The above rewrites object by combining two different options (Bread or Fish) into one 'Food' category.

Function for suppression

Now that we now how to write our own functions, we can write one to help us suppress data. First of all, we can demonstrate the suppression using functions from existing packages. At this point we should still have `student_results_aggregated` in our environment. We want to suppress any rows that have less than 5 students. One way to do this is shown below:

```
student_results_aggregated_suppressed <- student_results_aggregated %>%
  mutate(g1_mean = ifelse(students < 5, 'c', g1_mean),
         g2_mean = ifelse(students < 5, 'c', g2_mean),
         g3_mean = ifelse(students < 5, 'c', g3_mean),
         students = ifelse(students < 5, 'c', students)
         )
```

The above code takes the mean grade columns and uses an `ifelse()` statement *within* a `mutate()` statement. Here, it's saying for each of the mean columns, if the number of students is less than 5, then replace the entry in that mean grade column with a 'c', if not, then keep what is already in the column. Note that we *have** to do the students column last, because once we add 'c' to any column it becomes a character variable rather than numeric. Since the previous columns are based on the numeric value of the students, we must order it so that these mutations occur first, while the student column is still a numeric variable and therefore able to apply the `students < 5` logic.

While the above code works, the same issues arise as in the previous aggregation section where you can end up copying and pasting the same line many times depending on the size of your real-life data sets! We want to avoid this type of repetition where possible so that we reduce the possibility of making mistakes (like forgetting to update every line you have copied and pasted). So let's create a function to suppress counts less than 5. With an appropriate comment to explain, ass the below to your functions.R script.

```
suppress_counts <- function(column, count) {
  ifelse(count < 5, 'c', column)
}
```

The above function takes a column, and performs the `ifelse` statement from the previous example. Once you have created the function and can see it in your environment, you can rewrite the above example like this:

```
student_results_aggregated_suppressed <- student_results_aggregated %>%
  mutate(g1_mean = suppress_counts(g1_mean, students),
         g2_mean = suppress_counts(g2_mean, students),
         g3_mean = suppress_counts(g3_mean, students),
         students = suppress_counts(students, students))
```

However, this *still* includes some repetition we would rather avoid! therefore, we can take this one step further and make use of the `dplyr::across()` function. The `across()` function allows you to apply the same function to multiple columns - type `?dplyr::across` in the console to view the help page and more information.

We can rewrite the above in the following way:

```
student_results_aggregated_suppressed <- student_results_aggregated %>%
  mutate(across(c(g1_mean, g2_mean, g3_mean, students), ~ifelse(students < 5, 'c', .)))
```

So, we've just written the same thing in 3 different ways! While all three options produce the same output, the last example is the best for RAP, as it is the most concise and requires the least repetition.

Adding EES columns

Another common task that many analysts face is needing to add the required Explore Education Statistics (EES) columns formatted in the correct way. Using the online guidance, we can see that the following columns are required for EES: time_identifier, time_period, geographic_level, country_code (or geographic level equivalent), country_name (or geographic level equivalent).

When uploading data to EES, you should aim to include at least 3 years of data in EES where it exists in light of recent accessibility legislation. This example has 3 years of data, 2015, 2016, 2016. However, the time_identifier is not specified yet, and the year column is not correctly formatted!

Let's assume in this instance the time_identifier is *academic year*. We can easily add this column, and rename the existing year column using the following code:

```
student_results_aggregated_suppressed_EES <- student_results_aggregated_suppressed %>%
  mutate(time_identifier = 'Academic year') %>%
  rename(time_period = year)
```

We also want to add the required geography columns. In this case, we are going to use the 'school' column to get the required geography using `case_when()` within `mutate()`. Assume that the school 'GP' is in London, and the school 'MS' is in East England, and we want a *regional* geographic level (so we need region_name and region_code). We can use the following code:

```
student_results_aggregated_suppressed_EES <- student_results_aggregated_suppressed %>%
  mutate(time_identifier = 'Academic year', geographic_level = 'Regional',
         region_name = case_when(school == 'MS' ~ "East of England",
                                 school == 'GP' ~ 'London')) %>%
  rename(time_period = year)
```

The above code gets us the correct geographic level and region name, but not region code. We can use a *look-up table* to get the correct code. . .

joins & joining geographic codes via lookup tables

The dfe-analytical-services area on GitHub holds all publicly available code from DfE. The code behind the EES data screener is available here, and the screener repository contains a folder full of geographic lookup tables!

Today we will use the regions.csv file from this folder. We can pull the csv directly from the repo using the 'raw' URL: https://raw.githubusercontent.com/dfe-analytical-services/dfe-published-data-qa/master/data/regions.csv. Using this also means if the CSV is updated on GitHub, when you run your code it will automatically pull the latest version of the CSV! Adding appropriate headings and comments, pull in this lookup table using the following code:

```
regions_lookup <- read.csv("https://raw.githubusercontent.com/dfe-analytical-services/df
```

The lookup table should have appeared in the environment! Click on the name to open it, or use the code `View(regions_lookup)`. Now we can join the codes to our data using the region name using the `left_join()` function! We can 'pipe' joins in the exact same way we have piped summarise, group_by, mutate and rename statements. In the console, type `?join` to see the different types of join within the `dplyr` package!

Here we will use a left join because we want to keep all of the rows from our data, and only join on rows that match from the lookup table. We can use the following code to join the region code using region name and reorder the columns:

```
student_results_aggregated_suppressed_EES <- student_results_aggregated_suppressed %>%
  mutate(time_identifier = 'Academic year', geographic_level = 'Regional',
         region_name = case_when(school == 'MS' ~ "East of England",
                                 school == 'GP' ~ 'London')) %>%
  rename(time_period = year) %>%
  left_join(regions_lookup, by = c('region_name' = 'region_name')) %>%
  select(time_period, time_identifier, geographic_level, region_name, region_code, sex,
```

The data created should now look a lot like what you would normally upload to EES.

Reformatting data (wide -> long)

The data produced in the above section meets our EES data standards, and would pass all of the tests run by the EES data screener. However, it is not optimised in the best way

for users! Recent user testing found that when using the table tool, users preferred a 'long' data format rather than 'wide'. This instance is seen as 'wide' because we currently have 3 columns showing the average grade for three different assessments. The 'long' way to display this would be to have one column called 'assessment' and one column called 'mean_grade'. The 'assessment' column would indicate which assessment (1, 2 or 3) the grade in that row refers to.

You may be used to a version of this type of transformation in Excel using *pivot tables* for many reasons, not just for EES data. This can be easily done in R, which is better for RAP, and will also save you time and repetition in the future! In the `tidyr` package there are two functions that can be used to pivot tables - `pivot_longer()` and `pivot_wider()`. In this example, we want to use `pivot_longer()` to get the columns mentioned into a long format.

The code below will reformat the data:

```
long_data <- student_results_aggregated_suppressed_EES %>%
  pivot_longer(cols = ends_with("mean"),
               names_to = "assessment",
               names_pattern = "(\\d)+",
               values_to = "mean_grade",
               values_drop_na = TRUE)
```

View the data and take a look at how it is different now! `pivot_wider()` works in a similar way to go from long to wide.

Regular expressions

In the above code, names_pattern uses a *regular expression*. Regular expressions (often shortened to *regex*) are expressions that describe patterns in text, and are used in many coding languages other than just R (i.e. Python, C++, Javascript etc.). In the above, `"(\\d+)"` means 'pull out the numeric digits from the string'. In practice, this means in the new assessment column that is being created, rather than using the old column names of g1_mean, g2_mean and g3_mean as the entries within the assessment column, it will only pull the digit from those strings, which will be 1, 2 or 3.

Creating plots

Now that we have our data formatted nicely, lets imagine we want to create a chart. The most common package used to create charts in R is `ggplot2`. Install this package, add it to your library calls and snapshot your library for renv. You can view the package documentation online using this link: https://www.rdocumentation.org/packages/ggplot2/versions/3.4.1. While today we will create a simple grouped bar chart (which would preferably be created in EES if we were really publishing), the `ggplot2` package has a huge range of capabilities which are not yet available on EES, making it useful in those situations! Saving well-documented code used for plots and charts is also better practice than quickly creating charts in Excel, which can become lost, broken or distorted with no record of how it was initially created.

We want to create a grouped bar chart that compares the average grades of male and female students. `ggplot` works best with long data, which luckily, we now have! We will need to filter the data first of all to focus on one specific group. Try and take the 'long_data' we created previously (which should still exist in your environment) and create a subset called 'plot_data' with the following filters:

- time period: 2015
- age: 16
- region_name: London

Now we can use the 'plot_data' to create a bar chart using `ggplot2`. The following code creates a plot as described above:

```
ggplot(plot_data, aes(x = assessment, y = mean_grade)) +
  geom_bar(stat = 'identity', aes(fill = sex), position = "dodge")
```

Using online resources

An important skill when writing and developing code is knowing how to solve errors and find answers to your own questions. Since R is an open-source language that is free to use, the likelihood is any error or query you come across will have been asked by someone else before. The internet is full of forums and websites dedicated to R coding questions and guidance, making it an invaluable resource once you know how to find what you need!

Google questions and errors

The plot you have just created might not look the prettiest. Here, you are going to learn *how* to use Google for what you want/need in R! Top tips are to try and include the package name, keep it short, focusing on key words that might appear in other peoples similar questions, and always mention R somewhere so you find solutions in the right language!

Try and complete the following tasks using Google alone:

- Add a title to your plot
- Rename the axis labels
- Change the colours of the bars
- Format the Y-axis intervals
- Spell out 'Female' and 'Male' in the legend.

You are almost certainly going to create and cause errors while you experiment with the above. You can of course ask the workshop facilitators to help, but using Google to troubleshoot is one of the most valuable lessons you can learn for coding! Try and Google the error to see if someone else has already written an answer or solution online!

try chatGPT (something completely from scratch)

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

About this publication:

enquiries www.education.gov.uk/contactus
download www.gov.uk/government/publications

Follow us on Twitter: @educationgovuk

Like us on Facebook: facebook.com/educationgovuk