# Tidyverse for Journalists 2024

Jonathan Stoneman/Luuk Sengers

## R {1:3}

This version of our introduction was created to support our three sessions at Dataharvest 2024. Documentation and the files used in the session are available at the github repo - https://github.com/Stonepeople/DH24

In order to make the session easy to join without having to install a lot of software in a short time, we created a cloud version of the material needed to follow the sessions, complete with the necessary data. This project will remain active for the foreseeable future, and so you can continue to use R and the Tidyverse packages we have installed for you until you decide you want to install R and RStudio on your own hard drive.

The link to the cloud version of the project is

HERE (you will need to create a posit.cloud account, but you don't need to pay for it)

### Installing R and RStudio

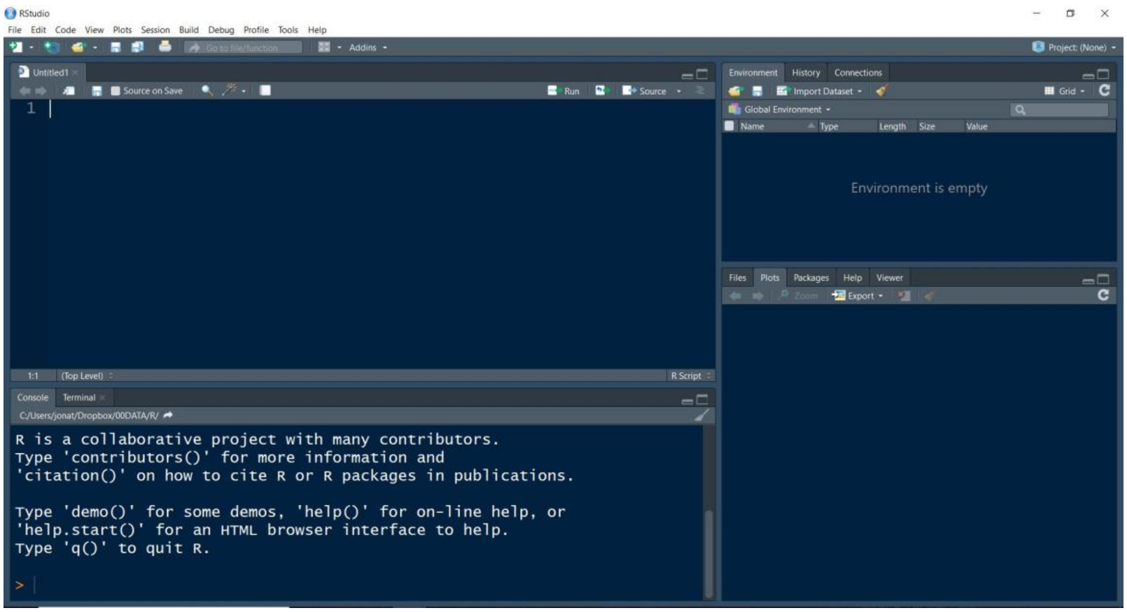
When you get to the point where you want to run R and RStudio on your own computer, you will need to

download and install R from: https://cloud.r-project.org

download and install R Studio from: https://posit.co/download/rstudio-desktop/

Why do you need RStudio? If R is the engine, then RStudio is the dashboard and controls - that is, you could run an engine by hand with the hood up, but you would probably prefer to run it from a nice user-friendly control panel, like RStudio!

## Your screen




The standard layout (background colour may be different - you can change yours with Tools>Global options >Appearance and choose one you like. This dark background is called Cobalt.

You will see 4 panes within your screen

- Top left is the script - you write instructions and comments in a script, which you can save and re-run at any time.
- Bottom left is the Console - that's where your instructions are carried out. You can write code in the Console directly, but it isn't saved in a way which is easy to re-use it. So most often you will click "Run" on a few lines of code, and watch them being carried out in the Console. Any error messages will appear there.
- Top right - is the Environment (and a couple of other tabs we don't need right now) That's where you dataframes will appear, along with any "values" you store - urls, or other "objects" you need to save in the temporary memory.
- Bottom right - several important tabs - the list of packages installed on your computer (or in the cloud project you're using), the files in your project, the help pages, and any plots you make.

## Projects

The word project sounds a bit grand, and self-important. In fact, keeping files and scripts together in a project is the easiest way to keep on top of everything you're doing: a project is simply a folder, where you keep together all the data, the scripts, the plots and anything else you need.

(When I started using R I made the mistake of thinking I didn't need a project - I was "just learning R" - I soon ended up with a folder full of files, scripts, datasets, and the odd plot…I couldn't remember what went with what. I soon realised that projects were the way to go!)

## Packages

A vital element of being an R user is access to the enormous range of packages which you can use to do various jobs (more than 20000 at the time of writing).

Packages are created by various R users, and, after testing and adoption by the R project, are documented and maintained by the community. Packages are, in effect, apps or mini-programs. Some do one specific job, others are built to carry out a range of tasks, such as importing data, cleaning it, making visualisations, etc.

If you look at your package pane (bottom right of screen) you will see two lists - the System Library (basic packages which come pre-installed in R and make it work, and carry out some frequent tasks) and the User Library (the packages that you have installed).

Some packages are recommended to you - like the Tidyverse, right now: you will come across others as you continue your R journey. Some packages rely on others to work - these are known as "dependencies".

When installing new packages, if you use the GUI method of adding new packages it's worth making sure the dependencies option is ticked. If you don't, then you will get warning messages about missing packages when you try to use them. It's not a big problem - you can install them anytime, it's just a little frustrating if it keeps happening when you're starting out.

We are going to use a group of packages known collectively as the "`tidyverse`". The tidyverse is a group of about 20 packages - and it's helpful to run them under one generic name instead of having to remember what task each one does, and loading it separately.

You install a package once, but you run it in every session when you need it: think of apps on a phone - you buy and install an app once but click on them to run them when you actually need.

## Writing our first lines of code

To get started, we now need to run the tidyverse, and then import some data to work with.

If you are working on your own computer, you first need to install the tidyverse. In the cloud session it's already installed, we just need to run it - using the command "`library()`".

The basic syntax of code is always the same - there's a function - in this case, library, followed by a pair of brackets(). Sometimes there's nothing between the brackets, but, as you will see, we often have to put details inside the brackets - such as the name of the dataframe or file we want to do something to, what we want to do, and more details.

To run some code, either put the cursor in the line you want to run and click on Run at the top of the pane, or hit CTRL (CMD on Mac) + ENTER.

At this point we are running just one line, but occasionally you may want to run a specific block of code, in which case you highlight the chosen block and hit CTRL + Enter, or the Run button.

```
library(tidyverse)
```

Running the command `library(tidyverse)`results in what looks like trouble, with mention of errors and conflicts - but don't worry, as long as you end up with an empty > then everything has run ok.

Now we need some data to look at. We are going to use some data supplied by the European Environment Agency - the European Pollutant Release and Transfer Reporting portal.

If you go to the bottom right hand pane, and select the files tab, you should see a file called "`releases_air_2023.csv`" - that's what we're going to import and work with.

*(You notice the name has underscores instead of gaps - R doesn't work well with spaces in file names or column names. We will keep it happy by using underscores.We've all got used to Windows and Mac OS allowing us to use anything we like as a filename. The operating system smooths it out for us, but languages like R won't put up with this habit!)*

To get the file into the Environment, where we can work with it, we need to use the `read_csv()` function. This is part of the readr package, which in turn is one of the tidyverse suite (now you see why we load the tidyverse instead of remembering all the individual packages we need to run!)

```
releases <- read_csv("releases_air_2023.csv")
```

If all goes well, and we don't have any errors, we should see a new object appear in the Environment in the top right pane - the line of code we just wrote and ran told R to give the name releases to whatever data it imported with `read_csv()`. We can see that it contains 340575 obs(ervations) - ie rows of data, and 17 variables (or columns).

### Finding out what's in your data

Now we can take a look at what is in the dataset - and there are several ways of doing so. Finding out what's in your data in R was, for me, the most disconcerting thing about moving to R after many years as a spreadsheet user. But once you get used to it, you will find there's nothing to worry about.

If our data were a spreadsheet, we'd probably select some columns and do some exploring. But in R, our relationship with the data is less "point and click" and more "ask focused questions using simple instructions".

Spreadsheet addicts could look at this dataset by typing and running `View(releases)`.That will show us something resembling a spreadsheet. In that view we can do simple sorting and filtering if we want to: just like old times!

But we're going to do it the code way - remember, one of our ultimate aims is to write code we, or a colleague, can run at any time in the future - by saving our instructions in a script.

Clicking on and around spreadsheets doesn't leave a record we can return to and reuse another time, we have to remember what we did and how we did it.

So let's look at the columns - to see what we've got

type into your script and run

```
names(releases)
```

Perhaps we want to see a list of the country names in the data -

```
unique(releases$countryName)
```

`unique` is the function that will take one copy of each entry in a column, and inside the () we put the name of the dataframe (`releases`) and the column we're interested in, with a `$` to separate the dataframe name from the column name

Now take a look at the names of the pollutants

How many are there? Use `n_distinct()` to count them

```
n_distinct(releases$pollutant)
```

You should get the answer 69.

How many countries? Try asking…

You should get the answer 32

There are two other quick summariser tools you can use to get a quick look at what's in your data set `summary()` and `glimpse()`

Try using them - as before, `releases` is the dataframe you want to look at. Take a look at the results - see what you get.

What do you find useful?

What doesn't help you?

Over time you will acquire your own set of habits for exploring your data. In this session we're going to answer various questions about the data without doing much to reshape it - restricting ourselves to counting, filtering and sorting the data to answer our initial questions.

So, we know how many countries there are in the dataset. How many facility names?

In theory the number of facility names should match the number of "FacilityInspire IDs (otherwise there are some dirty entries with variants of names making the computer read them differently). Do the numbers match?

What about finding out how many facilities there are in a particular country? We will need to use a filter. And for the first time, we are going to write two lines of code, but get them to run as one, by chaining them together with a so-called "pipe" - which looks like this `|>`(it may also look like this %>%). There is a shortcut to insert a pipe CTRL or CMD on a Mac + SHIFT + M (for "more"). At the moment, the two pipes are interchangeable. You can choose which you prefer in  Global Options > Code > Editing then tick "Use native pipe operator" if you want |> or leave it unticked if you want %>%.

Let's look at how many facilities there are in Belgium in our dataset:

```
Belg_facs <- releases |>
  filter(countryName == "Belgium")|>
  n_distinct(facilityName)
```

658

If we wanted to get a run down of how many facility names were listed for each country, just as with a spreadsheet, we could do it the boring way - rewrite the code we just used but change the countryName each time. Pretty boring. In a spreadsheet we would complete that analysis using a pivot table.

In the Tidyverse there is a function to do the equivalent: it's called `count()`

```
releases |> count(countryName,sort = T)
```

Maybe we want to break them down by what they do:

```
releases |> count(eprtrSectorName, sort = T)
```

What about breaking that list down by country as well?

```
releases |> count(countryName, eprtrSectorName, sort = T)
```

At this point, we are only looking at the number of facilities. But there is a column containing the amount of pollutant released by each facility - "emissions".

First, we might want to just check what kind of totals are in that column - the maximum, minimum, the mean, are the obvious ones. R will do that for us - in the function we need to tell it the name of the dataframe, and which variable we're interested in -

```
max(releases$emissions)

4.32e+10
```

If you want to analyse the emissions column a bit more try guessing the functions you will need in order to work out the minimum, the mean, the median, and the standard deviation.

Make the most of the way that RStudio autofills for you as you begin to type. Also, remember that you can ask for help by typing ? before the function - so `?min` will get you help on that, telling you what the defaults are, and what the syntax of any given function are - in other words, what the computer expects you to type, in what order.

If you prefer your numbers in raw format, not mathematical, you can reset the default setting for this R session. You need this command

```
options(scipen = 999)
```

Now, when you run the last instruction the max emission you get the same answer, but as a raw number -

```
43200000000
```

### Some advice about structuring your scripts

Running that line `options(scipen = 999)`is something we need to do every time we run a script - assuming we want to read numbers that way. So we ought to run that at the beginning, as part of our setup.

In this session we have been writing scripts as we go, asking R to perform certain tasks. If we keep the script from this session we can run it again and again, in exactly the same order.

When you start using scripts for your own projects, it makes sense to do things in a certain logical sequence.

The natural sequence is generally this -

1 - list the packages you need for this project

library(tidyverse) etc

(If you find later that you need extra packages, then you should put them with all the other package requirements (if you share your script with a colleague, it helps them to see from the beginning what packages they will need).

2 - import the data you need - in our case

```
releases <- read_csv("releases_air_2023.csv")
```

3 - cleaning, reshaping and wrangling steps.

4 - analysis

And finally

5 - some visualisations

It sounds simple, and it is - it is good discipline.

I have lost count of the number of scripts I've written where the order I did things is far from ideal, and when I come back to them days or months later, I waste time jumping up and down the script to find the task I need to do next. Eventually I get so annoyed that I stop and tidy the script into a better order: don't be like me - be organised!

## Exploring the data further

We can get quite a long way into the data using just one function: `count()`

We've already used it to count the facilities in the different countries. But we can do more than that.

We use a term within the function called wt - weighted total. If we ask it to count the facilities again, but this time take into account the quantity of pollution logged in the emissions column, we can quickly work out things like the largest pollutant in each country in each year

```
releases |> count(reportingYear, countryName, pollutant, wt = emissions)
```

If we want to sort in descending order of the amount of pollution we just add `sort = TRUE`

```
releases |> count(reportingYear, countryName, pollutant, wt = emissions, sort
= TRUE)
```

At this point, let's say we would like to keep that result, maybe we can use it later to make a graph - at the moment the table we just made is going to disappear at the end of the R session. Within the console it will move up, out of our immediate sight, as we run more code.

To make a storable copy we first make a copy as a dataframe within our project:

```
national_totals <- releases |>
 count(reportingYear, countryName, pollutant, wt = emissions, sort = TRUE)
```

That dataframe "national totals" exists only in our computer's short term memory. We need to save it by writing it to our hard drive -

```
write_csv(national_totals, "emissions_country_year.csv")
```

That .csv file will now work like any other file, we can email it to colleagues, and/or we can come back to it and import it next time we are working on this project. We can get back to it either by running the code needed to create, or just use `read_csv()`to import that specific file.

### More questions to ask the data

Have a go at asking some questions of this data. If you need some question ideas, try asking some of these:

- Which facility emitted the most CO2?
- In the Netherlands, which industrial sector emitted the most pollutants, and what was the biggest pollutant there?

- In the United Kingdom, what was the biggest single emission in 2021? (ie which facility, what pollutant, how much?)
- In Germany, what pollutant was the most emitted in 2021?

In the next session, R{2}, we will look at more complex ways of getting the data into a shape where it answers more questions.

## Session 2

In session 2 we assume you already know the basics - either you attended session 1, or you're acquainted with the basics of R and RStudio.

We will begin by importing a dataframe of emissions into the atmosphere, `"releases_air_2023.csv"`. We will call it `releases`.

In Session 1 we used some basic techniques to ask simple questions of the data. Just as we would with a spreadsheet program, we could learn quite a bit by sorting and filtering some of the columns: we could look at what is going on in different countries, in different years, and we could filter the emissions to look at emissions of specific pollutants.

But, just as we would do with a spreadsheet program, we can ask much more complex questions.

Every time we ask a question we have a choice - we can get the answers to appear in the console (bottom left pane on our screen), and we can re-run the relevant chunk of code to get them back at any time, or we can create a new dataframe which will sit in the Global Environment (top right pane). If we want to save the answer to a question as a file, we will need to make it a dataframe, and then "write" it as a csv file:

```
write_csv(that_frame, "name_of_saved_file.csv")
```

The standard syntax for creating a new dataframe is

```
new_dframe <- existing_dframe |>
  function(xyz)
    ## ie you do whatever you need to do, eg to answer your question:
    ## in this case, let's make a smaller version of the releases dataframe,
the first 1000 rows from the original dataframe, and a narrower set of
variables:

    releases_sm <- releases |>
  head(1000) |>
  select(reportingYear, countryName, eprtrSectorName, facilityName,
pollutant, emissions)
```

Now we have a 1000 row dataframe, with just 6 variables - it's called `releases_sm`

## Select()

The select() function allows us to pick whatever columns we want from the original dataframe, and put them in the order we want them. We can identify them by name or by the number of the column (you can run `names(releases)` to see the column numbers). In this case, the select() we ran is the same as

```
select(16, 1, 3, 7, 14, 15)
```

If you want a sequence of columns, 2 thru 4, for example, you can either type 2:4 or use the name of the first and last of the sequence - here that would be `EPRTRSectorCode:EPRTRAnnexIMainActivityCode` - so you can see why it might be quicker to type the column numbers!

## More on Filters

In the first session we used a filter to pick a country and concentrate on its emissions. The syntax is always `filter(the_dataframe, the_column_to_be_filtered the_condition_to_be_filtered)`

eg

```
filter(releases, countryName == "Netherlands")
```

we can negate that, and have everything except the Netherlands using "!" (aka "bang")

```
filter(releases, countryName != "Netherlands")
```

NB - if you are using the blocks of code as we've been doing the pipe (|>) means the whole block is dealing with the same dataframe, so you don't need to specify it in the code eg

```
releases |>
  filter(countryName == "Netherlands")
##is the same as
filter(releases, countryName == "Netherlands")
```

What if we want to filter for several countries in one go? There are two ways - as usual, one is a bit wordy, the other is labour-saving!

Let's go for the Benelux countries, Belgium, Netherlands, Luxembourg. First method -

```
Benelux_releases <- releases |>
  filter(countryName == "Belgium" | countryName == "Netherlands" |
countryName == "Luxembourg")
```

The "|" means OR. Note you have to re-specify `countryName ==` each time you use it.

That's quite a bit of typing, and we don't want to type that every time you wanted to filter by several countries. So there's another way, which can save time in the long run. First we define Benelux, then we use a special operator "`%in%`" to tell R we want any value in that group

```
benelux <- c("Belgium", "Netherlands", "Luxembourg")

Benelux_releases_2 <- releases |>
  filter(countryName %in% benelux)
```

We could use that same group to filter for all the countries EXCEPT Benelux:

```
Un_Benelux <- releases |>
  filter(!countryName %in% benelux)
```

Insert the negative bankg ("!") where you would say to yourself "not" - so "filter NOT the countryName in the Benelux group"!

There's another way of filtering which is even more useful.

R is very literal - we have had to specify precisely what we are looking for, inside double quotes - so "Belgium", not "belgium" or "belg" and if it's a name like "United Kingdom" we must use the same number of spaces between the words as there are in the countryName column. (And if anyone inputting the data had happened to put two spaces in the name, our filter will not find those entries).

But sometimes we need to filter for part of a word, or a short phrase that's in a longer sentence. For that, we need to use a function `str_detect` within the filter instruction.

The syntax is `filter(thedataframe, str_detect(thecolumnname, "the string we are looking for"))` – (remember to use enough closing brackets!)

So we could use `str_detect` to find the UK, by just searching for United in the countryName - filter(releases, str_detect(countryName, "United") but it's much more useful in the following way. Let's look for anything involving "manufacturing" in the column called EPRTRAnnexIMainActivityLabel:

```
manuf_releases <- releases |>
  filter(str_detect(EPRTRAnnexIMainActivityLabel, "manufactur"))
```

Why didn't we put the whole word? In this case "manufactur" will find both "manufacturing" and "manufacture". In fact, as we now see "manufacture" is the only variant in this data.

Now look for "ferrous metals" in the data….

```
ferrous_releases <- releases |>
  filter(str_detect(EPRTRAnnexIMainActivityLabel, "ferrous metals"))
```

Now there's a problem - that finds both "ferrous" and "non-ferrous". So we could insert an extra line to filter that out

```
ferrous_releases <- releases |>
  filter(str_detect(EPRTRAnnexIMainActivityLabel, "ferrous metals")) |>
filter(!str_detect(EPRTRAnnexIMainActivityLabel, "non-ferrous"))
```

(It's probably worth saying that `str_detect()` makes use of Regular Expressions - so you can do things like looking for any numbers in a string, or specific strings such as words beginning with B, or not with X. We don't have time or space for that now, but it may come in useful later)

There's a slight complication with str_detect, which is also worth mentioning at this stage. You can't use the `%in%` function to filter for a group of words or strings. You have to use a different syntax entirely. Let's look for "glass" or "brick" in the MainActivityLabel column:

First we make a search group as we did for Benelux:

```
glass_or_brick <- c("glass", "brick")
```

To ask R to find any entry containing either glass or brick, our code will look a little more complicated (sorry!)

```
glass_brick_releases <- releases |>
  filter(str_detect(EPRTRAnnexIMainActivityLabel, paste(glass_or_brick,
collapse = "|")))
```

What's happening here? We've used an extra function, `paste` , to insert the words in our little list called glass_or_brick, but then we need to tell R that it's either one or the other, by using the "|" (as we did earlier for Benelux). The `collapse` = tells R that the list, however long it is, collapses the commas between the search terms into the either/or of the so-called pipe "|".

## Reshaping data

So far we've been filtering data. Sometimes we want to reshape it, and use something resembling what we may already know as a pivot table: a way of grouping rows of data together, and reshaping the original data to help us get answers to our questions. In a point and click program like Excel or googlesheets what we're about to do is done for you. In R it is a little more typing, but ultimately much more flexible.

To create the equivalent of a pivot table we need three lines of code, in the same sequence every time.

Let's work out the total emissions by country in each year (this will add up ALL the emissions, so it's not really very meaningful, but it's just an exercise, and we can refine it to make it more meaningful shortly!)

We need to group the data by reportingYear and countryName, and then add up the emissions by those same groups -

```
annual_emissions <- releases |>
  group_by(reportingYear, countryName) |>
  summarise(total_emissions = sum(emissions)) |>
  arrange(reportingYear)
```

This has done what we said we wanted to do. At this point we just want to see each year in order. But we might reasonably want to see which country has the all time record for emissions….

```
annual_emissions <- releases |>
  group_by(reportingYear, countryName) |>
  summarise(total_emissions = sum(emissions)) |>
  arrange(desc(total_emissions))
```

NB - it's really important to note that the `summarise()` function is key to building your new table. If you don't summarise the data in some way, the grouping of variables just won't appear - it's a bit like building a pivot table: you can put variables in rows and columns, but if you don't add a variable to the values part of the table, it stays blank.

As we said, this number, impressive though it is, is a pretty meaningless combination of every pollutant released into the air in each country, each year. Let's look at CO2 releases specifically. We will filter the data before we group it - using `str_detect` to find every reference to CO2:

```
annual_CO2 <- releases |>
  filter(str_detect(pollutant, "CO2")) |>
  group_by(reportingYear, countryName) |>
  summarise(total_emissions = sum(emissions)) |>
  arrange(desc(total_emissions))
```

Finally in this section, let's see which industrial sectors are emitting the CO2 as well - we need to add another variable to the group by instruction:

```
annual_CO2 <- releases |>
  filter(str_detect(pollutant, "CO2")) |>
  group_by(reportingYear, countryName, eprtrSectorName) |>
  summarise(total_emissions = sum(emissions)) |>
  arrange(desc(total_emissions))
```

So far we've looked at releases into the air. There is of course a separate dataset for releases into water. Let's take a look at it:

```
releases_water <- read_csv("releases_water_2023.csv")
```

We see that it's a wider table than the releases into the air data, let's see what the 31 variables are: `names(releases_water)`

For some reason the European Environment Agency has organised the data in a completely different way - each year has its own column with the emissions for that year recorded in it.

The air data has "reportingYear" as a single column.

We want the water release data in the same format as the air release data. If we were to try to combine these by hand (eg copying and pasting in a spreadsheet) it would take many hours. But R has a function to look after exactly this problem. It's called `pivot_longer()`

because the action of taking a wide dataframe and making it narrower (putting all years in one column) will inevitably result in a longer dataframe.

We need to tell R which columns to combine, what to call the new column and what to put in it. Let's make a new table based on releases_water, we call the new year column the same as in the air dataset "reportingYear", and put the values in a column called "emissions" so the two dataframes will look similar:

```
releases_water_long <- releases_water |>
  pivot_longer(16:31, names_to = "reportingYear", values_to = "emissions")
```

This produces a dataframe of 1 277 968 rows and 17 variables from one of 79873 rows and 31 variables. Although it's hard to work with so many rows in a spreadsheet, it's much easier in R.

Now - it would be nice to combine the two datasets. If we were to do this in spreadsheets, it would mean making sure the two sheets had the same columns in the same order. In R they just need to have the same names. There are 17 in each case, and we can check the column names with the names function…

We will use a function called `rbind()` - we will create a new big dataframe called combined, and pass the names of the two frames to tell it what to work with.

```
combined <- rbind(releases, releases_water_long)
```

### Learning about functions - and getting help

By the way - as part of building your knowledge of R - if we explain that `rbind` means "bind rows" you might guess that there's also a function `cbind` - "bind columns". rbind joins dataframes vertically - adding more rows. On the other hand, cbind combines dataframes horizontally. With all functions, if you want to read more about them, type `?nameoffunction` - so type ?`rbind` in the console and the relevant help pages will open in the pane, bottom right.

The help pages are all formatted in the same way - starting with a Description, Usage (how the function works), Arguments - what goes between the brackets, Details - more narrative on how it works. This section is usually in quite technical language . It's followed by References, but at the bottom comes the most useful section - Usage, where you can see examples at work and in context.

## Joining dataframes to enhance your data

We joined a couple of dataframes which were essentially the same shape by adding rows. Another frequently used method of adding detail to your data is by joining two datasets, using at least one column as a key to match records together.

To show this in action, we're going to add in the surface areas of countries, which will in turn allow us to work out the amount of $CO_2$ per square kilometre.

First we need to make a dataframe showing CO2 emissions in each country. To make the data a managable size, and for a bit of practice let's take the top 5 countries, and look at pollution in the last 4 recorded years, 2019-2022.

```
top_5_CO2 <- releases |>
  filter(reportingYear >= 2019) |>
  filter(str_detect(pollutant, "CO2")) |>
  group_by(reportingYear, countryName) |>
  summarise(total_emissions = sum(emissions)) |>
  arrange(desc(total_emissions)) |>
  slice_head(n = 5)
```

The function `slice_head()` has taken the top 5 countries in each year for CO2 emissions. (If we just took the top 5 readings, we would almost certainly find the same one or two countries coming top every year).

Now let's import and look at the areas.csv file

```
areas <- read_csv("areas.csv")
```

It's important for joining purposes that the column names match. This will also give us the chance to demonstrate a new function - rename().

```
areas <- areas |>
  rename(countryName = country)
```

In the rename() function the format is always - newcolumnname = oldcolumnname.

We don't need all the columns in the areas, so we can use select() to choose just countryName and tot_area_sq_km

```
areas <- areas |>
  select(countryName, sq_km = tot_area_km_sq)
```

In doing that, notice how we also renamed "tot_area_km_sq" while we selected it. (Yes - we could have done the whole operation in one go, but we wanted to show the `rename()` function).

Now to the join. There are several options: they're all explained in the tidyverse documentation https://dplyr.tidyverse.org/reference/mutate-joins.html

We will use an inner join - which only keeps rows from the first dataframe we list which are matched in the second dataframe. So:

```
top_5_CO2_with_area <- inner_join(top_5_CO2, areas, by = "countryName")
```

Now - we can calculate the CO2 per square km. We need to use `mutate()` which allows us to do calculations with values from different columns. In this case we divide the total_emissions by the sq_km.

```
top_5_CO2_with_area <- top_5_CO2_with_area |>
  mutate(kg_per_sq_km = total_emissions / sq_km)
```

For the purposes of this session we broke the process down into steps. But the beauty of R scripting is that we can run the whole operation in one block of code - importing the areas, selecting, renaming the columns and joining the two together.

The number of operations you can run in one block depends mainly on your ability to debug any errors - when you first start, it's wise to take relatively small steps, but as you get more experienced, you begin to see what operations can be chained together in a series of "pipes" - `%>%` or `|>`.

Writing R code becomes like writing any language - you start with simple sentences, but learn how to put a lot of ideas into one long sentence that flows better than several shorter ones.

## Session 3

In session 3 we assume you already know the basics - either you attended session 1, or you're acquainted with the basics of R and RStudio.

We will be looking at how to make visualisations in R using the package ggplot2. We will also take a look at how we can use generative AI such as chatGPT to help write code.

The data we will be using to build visualisations is based on the releases into air - the data frame we imported as releases in Session 1.

If you have just joined, or if you don't have the data ready you should run this script block

```
releases <- read_csv("releases_air_2023.csv")

annual_top_5 <- releases |>
  filter(reportingYear >=2019) |>
  group_by(reportingYear,countryName,pollutant, eprtrSectorName ) |>
  summarise(annual_emissions = sum(emissions)) |>
  arrange(reportingYear) |>
  slice_head(n = 5)
```

Creating annual_top_5 gives us some useful data to try visualising.

ggplot2, now referred to mainly as ggplot was the original package in what is now the tidyverse suite of packages.

gg stands for Grammar of Graphics: a logical way of breaking down the process of turning data into a graphic. Put simply it involves answering three key questions:

- which data do I want to display?
- what variables go on the x axis and the y axis, are any other variables represented by shapes or colours?
- what type of graph do I want to use? (bar, line, pie, etc)

As we build up the R code, we will answer those questions in sequence. Of course, there are further questions around the design of the graph - colour choices, labels, tooltips, etc. But let's get the basics out of the way first:

```
ggplot(annual_top_5, aes(x = reportingYear,y = annual_emissions, fill =
eprtrSectorName)) +
  geom_col()
```

**NB - we have stopped ending lines with the pipe, and are now using + instead. This is a ggplot thing - going back to the early days of the tidyverse. The %>% came later, and the |> later still. So we just have to live with this weird little quirk. Sorry!**

What do we have? It's not great as graphics go - the legend is bigger than the graph!

Let's make it a bit easier to read by making lots of little graphs - one extra line:

```
ggplot(annual_top_5, aes(x = reportingYear,y = annual_emissions, fill =
eprtrSectorName)) +
  geom_col() +
  facet_wrap(vars(eprtrSectorName))
```

Still not great. We might want to concentrate on fewer sectors, and since we have mini-graphs for the sectors, we could use the colour for another variable - countryName, for example.

```
ggplot(annual_top_5, aes(x = reportingYear,y = annual_emissions, fill =
countryName)) +
  geom_col() +
  facet_wrap(vars(eprtrSectorName))
```

Still a mess - but it's easy to reshape the graph. Let's concentrate on the energy sector only.

```
annual_top_5 |> filter(eprtrSectorName == "Energy sector") |>
  ggplot(aes(x = reportingYear,y = annual_emissions)) +
  geom_col() +
  facet_wrap(vars(countryName))
```

At this point, we don't have a graphic we can show our audience - but it is helping us to see what's worth concentrating on. We could, for example, hone in on the "busier" countries in the plot - Germany, Italy, Netherlands, Poland, Spain.

```
chosen_countries <- c("Germany", "Italy", "Netherlands", "Poland", "Spain")

annual_top_5 |> filter(eprtrSectorName == "Energy sector") |>
  filter(countryName %in% chosen_countries) |>
  ggplot(aes(x = reportingYear,y = annual_emissions)) +
  geom_col() +
  facet_wrap(vars(countryName))
```

And so we go on - reusing the code, focusing on what seems important - adding a line here, removing a variable there.

We should at some point add a title and subtitle

```
chosen_countries <- c("Germany", "Italy", "Netherlands", "Poland", "Spain")

annual_top_5 |> filter(eprtrSectorName == "Energy sector") |>
  filter(countryName %in% chosen_countries) |>
  ggplot(aes(x = reportingYear,y = annual_emissions)) +
  geom_col() +
  facet_wrap(vars(countryName)) +
  labs(title = "Most polluting Energy Sectors 2019-2022", subtitle = "Top 5 
countries by total emissions, all pollutants")
```

There is a lot to take in, but ggplot is a very flexible tool once you get used to it. As with so much R, you need to play around and learn as you go. We could spend hours just on the options available here - but we don't have that much time.

If you want to do more work with ggplot there are lots of online tutorials available - Cedric Scherer's "Beautiful Plotting in R"is one of the best.

See also notes on other sources of help at the end of this handout.

There's a lot to learn, as there is with any new language. The key thing to remember is that unlike a point-and-click tool like googlesheets or Excel, your script remains a record of what you did.

Even if you come back to a script after a long break, it will still work (unlike trying to remember what you did to your data in a spreadsheet!) If you see something you like - a great visualisation for example - you can usually get a copy of the script used, and then adapt it to data you're working on.

## Getting AI to help

Nowadays you can enlist the help of AI to help write your scripts. At the very beginning of your R journey, this could save time, but might also make life a little harder – since you are not yet well enough acquainted with R to see what's good and what's bad.

The trick is to understand enough of the R code to know what you need to change to adapt it to your needs.

Wwe can ask, say ChatGPT, to write code for us and save time. But we then need to be able to spot any errors in the code, and change them as appropriate.

There is a useful mnemonic for getting AI tools to suggest code -

**S2C** - keep it Simple, be Specific, and use Comments (the "non-code" preceded by a # so the computer ignores it)

So - let's ask ChatGPT for help with a visualisation.

*Question I have a dataframe showing the top 5 countries emitting pollutants 2019-2022. The columns are [1] "reportingYear" "countryName" "pollutant" "eprtrSectorName" "emissions". I* *need a script using ggplot and the tidyverse to make a visualisation showing each country's total emissions over time*

Answer: To visualize each country's total emissions over time using ggplot and the tidyverse, you can follow these steps:

1. Filter the dataframe to include only the top 5 emitting countries.
2. Group the data by country and reporting year to calculate the total emissions for each country in each year.
3. Create a line plot with ggplot, mapping reporting year on the x-axis, total emissions on the y-axis, and color by country.

Here's a script that achieves this:

```r
library(ggplot2)
library(dplyr)

# Assuming your dataframe is named "releases"
# Convert reportingYear to a factor to maintain chronological order
releases$reportingYear <- factor(releases$reportingYear)

# Calculate total emissions for each country and year
total_emissions <- releases %>%
  group_by(reportingYear, countryName) %>%
  summarise(total_emissions = sum(emissions))

# Plotting
ggplot(total_emissions, aes(x = reportingYear, y = total_emissions, color =
countryName)) +
  geom_line() +
  labs(title = "Total Emissions Over Time",
       x = "Reporting Year",
       y = "Total Emissions",
       color = "Country") +
  theme_minimal()

library(ggplot2)
library(dplyr)

# Assuming your dataframe is named df
# First, convert reportingYear to a factor to maintain chronological order
df$reportingYear <- factor(df$reportingYear)

# Then, calculate total emissions for each country and year
total_emissions <- df %>%
  group_by(reportingYear, countryName) %>%
  summarise(total_emissions = sum(emissions))

# Now, create the plot
```

```
ggplot(total_emissions, aes(x = reportingYear, y = total_emissions, color =
countryName)) +
  geom_line() +
  labs(title = "Total Emissions Over Time",
       x = "Year",
       y = "Total Emissions",
       color = "Country") +
  theme_minimal()
```

I responded and said I'd prefer a bar plot - and chatGPT came back with this

```
library(ggplot2)
library(dplyr)

# Assuming your dataframe is named df
# If not, replace df with the actual name of your dataframe

# Step 1: Filter the dataframe to include only the top 5 emitting countries
top_5_countries <- df %>%
  group_by(countryName) %>%
  summarise(total_emissions = sum(annual_emissions)) %>%
  top_n(5, total_emissions) %>%
  pull(countryName)

df_top_5 <- df %>%
  filter(countryName %in% top_5_countries)

# Step 2: Group the data by country and reporting year to calculate total
emissions
df_total_emissions <- df_top_5 %>%
  group_by(countryName, reportingYear) %>%
  summarise(total_emissions = sum(annual_emissions))

# Step 3: Create a bar plot
ggplot(df_total_emissions, aes(x = reportingYear, y = total_emissions, fill =
countryName)) +
  geom_bar(stat = "identity", position = "stack") +
  labs(title = "Total Emissions Over Time",
       x = "Reporting Year",
       y = "Total Emissions") +
  theme_minimal()
```

First test - does the script work?

I forgot to specify top 5 countries for emissions, so - amended code comes back:

```
library(ggplot2)
library(dplyr)

# Assuming your dataframe is named "releases"
# Convert reportingYear to a factor to maintain chronological order
releases$reportingYear <- factor(releases$reportingYear)

# Calculate total emissions for each country and year
total_emissions <- releases %>%
  group_by(reportingYear, countryName) %>%
  summarise(total_emissions = sum(emissions)) %>%
  arrange(desc(total_emissions)) %>%
  filter(countryName %in% unique(head(countryName, 5)))

# Plotting
ggplot(total_emissions, aes(x = reportingYear, y = total_emissions, color =
countryName)) +
  geom_line() +
  labs(title = "Total Emissions Over Time for Top 5 Polluting Countries",
       x = "Reporting Year",
       y = "Total Emissions",
       color = "Country") +
  theme_minimal()
```

Does this work? Not quite: so the questions are

- Can you improve on the code ChatGPT supplies ?
- Which is harder - writing the code, or correcting the code?
  - Do you want to ask ChatGPT to correct it, or make the corrections yourself?

## Integrating AI tools into RStudio

We used ChatGPT for these examples, mainly because ChatGPT 3.5 is free, and its code blocks are easy to copy and paste into a script. Perplexity is similarly free to use. The challenge lies in giving either LLM a good set of prompts. But like LLM itself, you will learn what works, and you will learn good and bad coding as you do so.

There are (currently) two ways of integrating AI more closely into RStudio. Both cost money - though there you can test the paid version for free before committing money to it.

RStudio now integrates with GitHub Co-pilot ($100 a year, free to try for 90 days). When running Co-pilot you can write what you want to do as a comment, and when you pause, a ghost line of code appears: to accept it, you hit the Tab key and the code is added to your script. Like any code, it doesn't get run until you run it.

- I have tried this with mixed results. Some of the code includes methods I would not have thought of, but sometimes Co-pilot overcomplicates its approach. In one case, when I asked it to sort some data by population density, it failed to "see" that there was a column with that name, and tried to calculate population density using the population column, which did exist, and dividing it by the area column, which did not!
- On another occasion I paused for quite a long time, and unprompted Co-pilot proposed making a complex plot, which wasn't something I was planning on making at that point! But, as I said, you don't have to run the code it offers - you can even reject it by typing over the ghost code.

  The other option at the time of writing is to use an R package - chattr. This requires some setting up. You can read about it, and its benefits and pitfalls in thisr-blogger post

## Getting further help - resources

If you have liked what you have learned so far, you will want to get more help. The best place to start is R4ds.hadley.nz - a free, online version of the 2nd edition of Hadley Wickham's guide to the Tidyverse "R for Data Science". It contains code and exercises and is well worth spending time with. The first edition is still worth looking at - and the authors make their points in different ways, which is always interesting.

#tidytuesday is well worth following on X/twitter - every Tuesday a group called RforDataScience puts out a dataset for the community to play with. Over the next day or two people tweet their visualisations - it's a great way of seeing how others can use the same data.

More usefully still, there are screencasts on youtube (same search term) showing how to analyse the data - the best are the ones by David Robinson (@drob) . This is a whole series of R masterclasses! Oscar Baruffa turned the indexed library of screencasts into a user-friendly, searchable website - rscreencasts.com

r-bloggers.com - a searchable site, which also produces a daily newsletter email Youtube (search for package by name)

Rstudio.com for cheatsheets and news about RStudio

Stackoverflow.com - searchable knowledge base on most code languages, including R.

Rseek.org - a search engine adapted to concentrate on R-related questions

Rdrr.io - a really well-organised repository of documentation about every R package

The Big Book of R (curated by Oscar Baruffa) links to a huge range of free resources to help you learn R. It is well-maintained and contains links to a wealth of amazing material

Rforjournalists.com is yet another excellent, free, resource.

And The Tidyverse Cookbook still in development shows you how to do the things you need to do.

You can do paid courses, sometimes with a free introductory period, at:

- Datacamp
- Udacity
- Coursera
- Lynda
- Codeacademy - learn.r is part of Codeacademy's excellent offer

As you dig into graphics with ggplot a whole new world opens up. Cedric Scherer's tutorials are superb. And don't miss the "BBC Visual and Data Journalism cookbook for R graphics", which includes code and a BBC-created R package for publication-quality ggplot2 graphics: https://bbc.github.io/rcookbook

Feel free to contact either of us with any questions you may have:

Jonathan Stoneman – Jonathan.stoneman@gmail.com

Luuk Sengers – info@luuksengers.com