EpidemiaR Introductory Materials

Justin Millar Amir Siraj

2024-06-20

Table of contents

Getting started

This book contains background material for general R basics, as well as some of the packages and functions that we will use during the EpidemiaR training lessons.

> ⚠ Check your R version!
>
> Some of the training material will use a special operator called the "native pipe" (which looks like this: `|>`). This function was first included in R version 4.1, which was released in May 2021.
> If you haven't updated your R in while then it may be time to update. You can check what your current version is by running `R.version` in the console.

Installing packages

The training lessons will use several R packages. We will discuss these in more detail in the following sections, but it is a good idea to get everything installed and downloaded before we dive in. Note that depending on your internet connection this may take some time.

Use this code in R to install the packages:

```
# Installing packages
install.packages(
  c("dplyr","knitr","lubridate","parallel","readr","readxl","tidyr","tinytex",
    "tools","tidyverse","janitor","writexl","sf","ISOweek"))
```

Additional materials and resources:

- WIP!

Part I

Introduction to R and Rstudio and basic functions

Introduction to RStudio

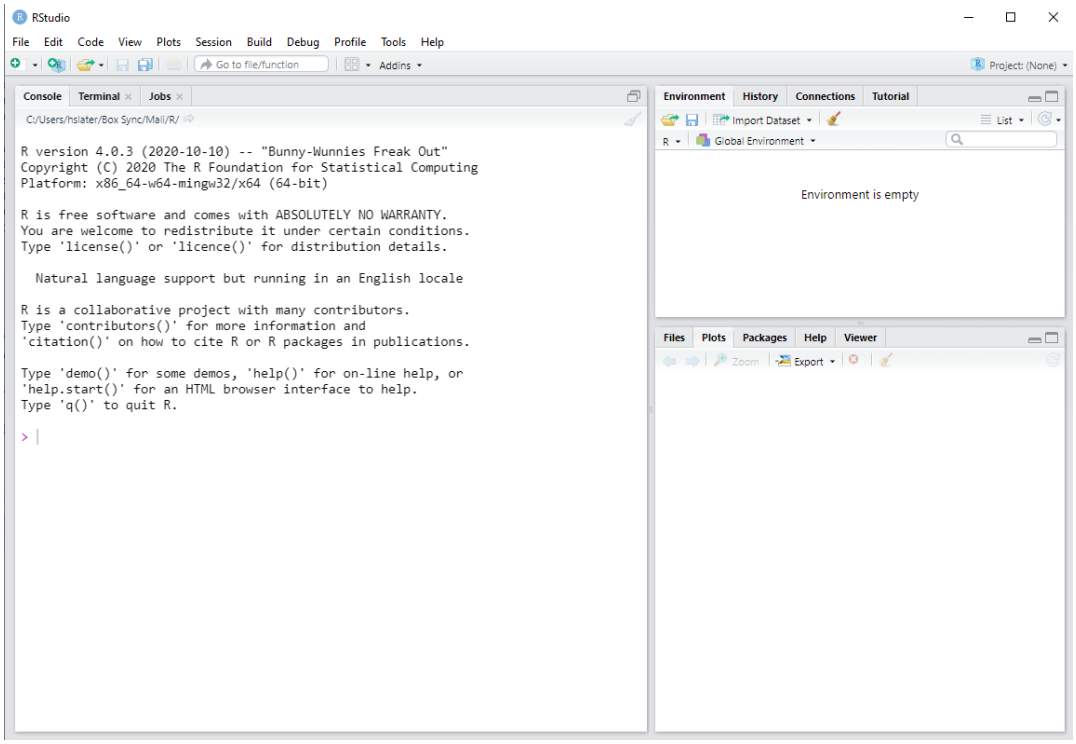
R is a statistical and graphical software package, and is the very commonly used in many disciplines, including data science, statistics and the environmental and biological sciences. The great strengths of R for the research community are:

- It is free, which means anyone can use it.
- It is open source, which means you can inspect and modify the code within in.
- It is a community, so researchers around the world have developed code for particular applications that they have made freely available. These are published as 'packages' which can be downloaded with just a couple of lines of code.
- Because of all the community-developed code, there are functions available in R for pretty much every data analysis, statistical method or model, graphic or chart you could ever need. Once you understand how to use R, you will be able to access these.
- There is an incredible amount of help to be found online for most problems (often on StackOverflow and Twitter #rstats).

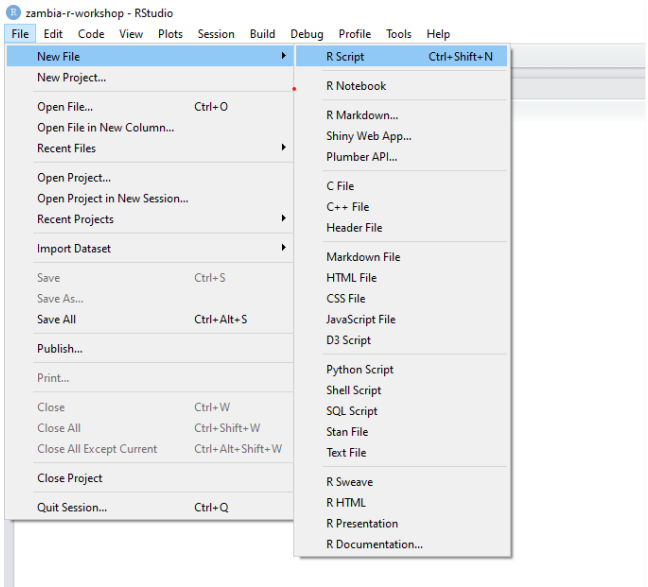
By the end of this session we will have covered

1. Opening up Rstudio
2. Setting up a project in Rstudio
3. Exploring R, finding help, debugging and writing comments
4. Different data types
5. How to calculate basic operations on vectors
6. How to use dataframes
7. What packages are and how to install and use them
8. How to read in external data sources

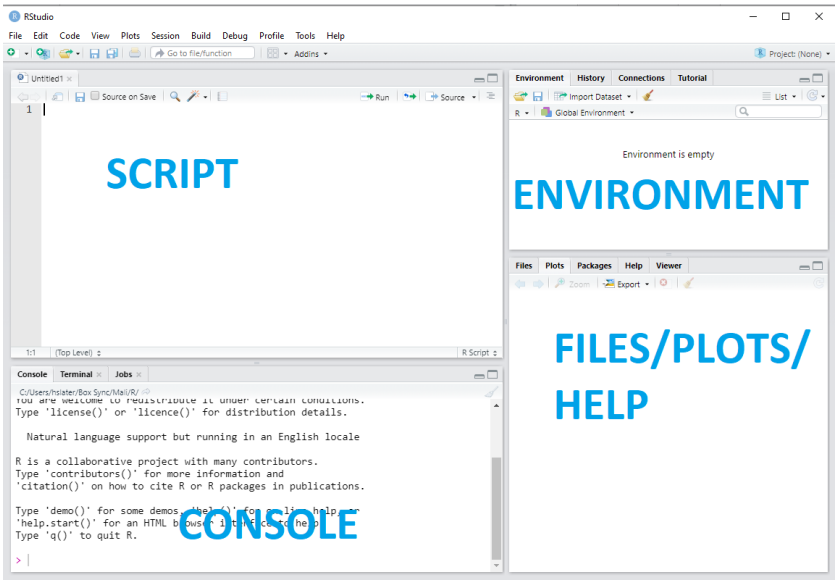
Hopefully you have followed the tutorial we shared and have RStudio downloaded and installed on your computer If you open up Rstudio, you will have 3 panels that look like this



The first thing you want to do is open up a new **R script**



Your Rstudio will now have 4 panels



These screens are:

Top left: An "R Script" window – this is essentially a text editor where you can write code and commands to be run in the console window. You can edit and correct your work here, and save it so you can use it again. I suggest saving regularly, and keep all your successful code here so you can keep working. Try to use informative script names to help your future self, for example you could save this week's work as "R_training_day1".

Bottom left: This is the "R Console" – this is where the processing is done. You should typically write script code in the "R Script" window and press Ctrl+Enter (or click "Run") to run them in the console. You may sometimes want to type directly in the console to quickly check that new object looks how you expect them to look or run simple checks that you don't want to revisit later

Top right: The "Environment" tab here shows all the data you've loaded and variables/objects you've created.

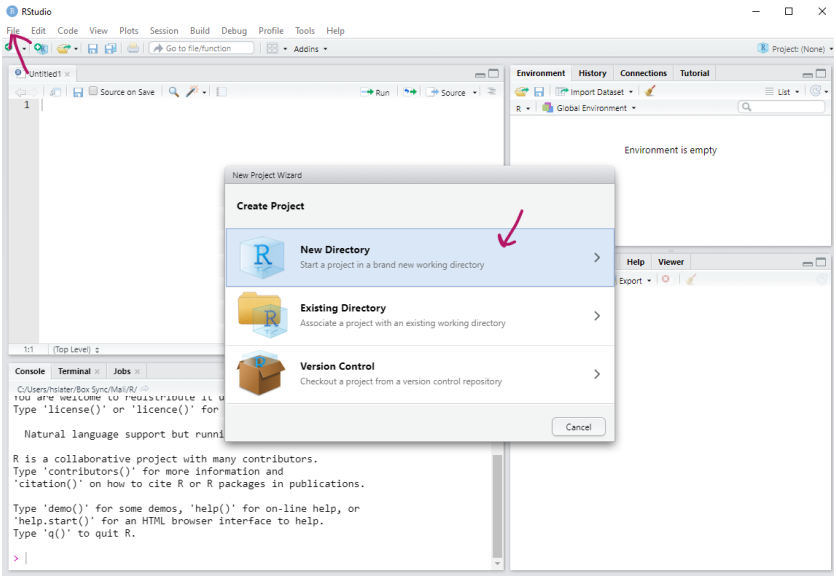
Bottom right: Plots you create will show here. You can also use R Help here: type a function you're using into the search box and you will get information on its inputs ("Arguments") and outputs ("Value"), or in the console write a '?' then the function name, i.e. '?mean' to understand how to use the 'mean' function

Setting up a project in RStudio

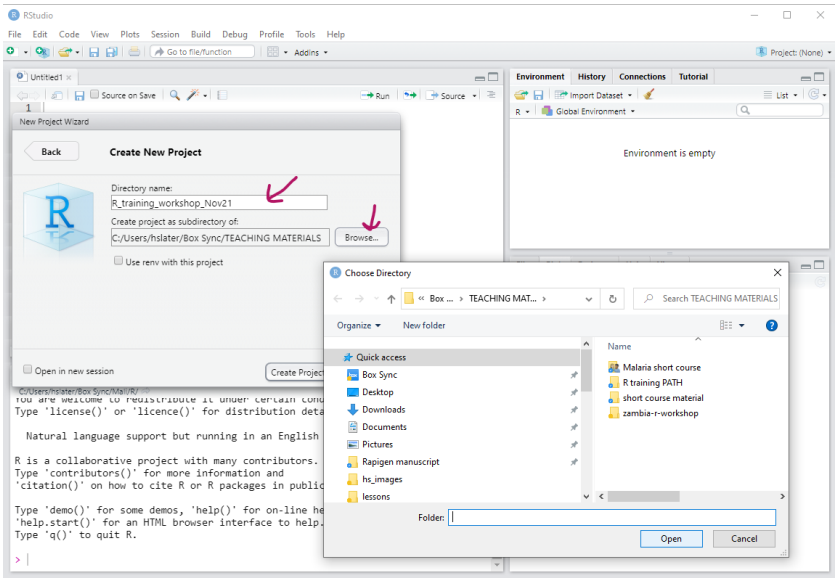
Projects are a really neat way to work in Rstudio – it means that all the inputs (i.e. data, scripts) and outputs (i.e. plots, summary tables) are stored in the same place. It also means other people can use the same project and rerun our analyses without having to define new

filepaths (which can be a pain!) When starting a new project, it is a really good practice to set up a project and associated file structure

Firstly go to **file -> new project**, then select **new directory**, then **new project**



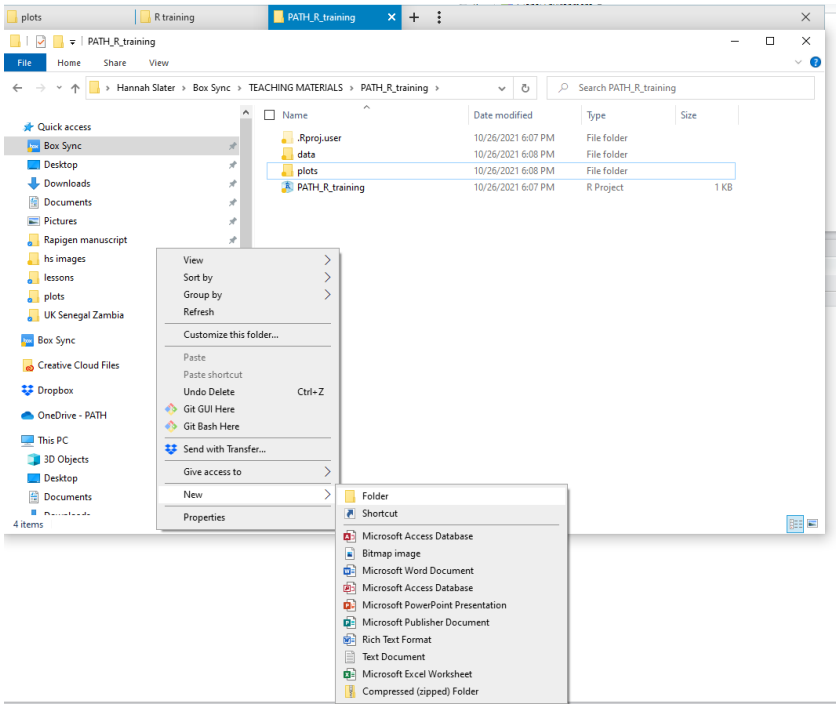
Define your directory (folder) name (I'd suggest something like **R_training_workshop_Sept23**) Then use the **browse** option to save this folder somewhere sensible for you (I tend to save things like this in a training folder on my box drive)



Now you have a new project!

The next step is to open up a file explorer window and locate this folder

Manually create three new subfolders called **data**, **scripts** and **plots** – everything we use and create during this workshop will be stored here for easy reference and will be able to be reproduced by other or yourself on a different computer



Exploring R, finding help, debugging and writing comments

We can now start using R – the simplest thing to do is just to use R as a basic calculator. You can write equations in the script and run them in the console to do simple calculations

```
20 * 10.5
```

```
[1] 210
```

```
17 + 23 - 7
```

```
[1] 33
```

There are also a wide range of mathematical functions such as log(), sqrt(), exp()

```
sqrt(25)
```

```
[1] 5
```

If you don't know what an inbuilt function does, type a '?' before its name, and the help page will appear on the right hand side of your screen:

```
?sqrt
```

A note on errors – these happen *ALL THE TIME!* A large part of learning to code is learning how to spot and fix your errors – this is called debugging (a great hobby for malaria researchers!)

For example, try the following - why do they not work?

```
sqrt(a)
sqrt("a")
SQRT(25)
sqrt(25))
```

If your code isn't working, the first few things to check are:

- Have you spelled the variable names and function names correctly?
- Do the objects you're working with exist in your working environment?
- Do you have the '>' appearing in the console window? If not something didn't run properly so press **Esc** a few times until it reappears

If your code is still not working, please grab a trainer and we will help you debug!

A note on commenting code - it is really good practice to comment your code - this means writing brief descriptions of what you've done.

By adding the '#' before a line of code, it will not be run in your console

```
# write in a description here so you can remember what you did in the future!
# here I am using the formula for the area of a circle pi * r^2
area_of_cirlce <- pi * 4 ^2
```

You can also comment out (i.e. add a # before them) lines of code that are maybe wrong or not needed in an analysis before you decide if you need to actually delete them or not. This is good practice, because sometimes we are too hasty in deleting lines of code that we might actually need!

Data types

R can store and organize data, parameters, and variables of lots of different types – we will now explore a few of them. To start, we need to create a new variable with a name. The we can assign values or data to that variable using `<-` . Here are some examples of different types of information that can be store in a variable.

Numbers

Single numbers can be given names like this

```
my_first_variable <- 20
my_first_variable
```

```
[1] 20
```

It is important to think about how to name variables – do you want to be quick, or do you want your code to make sense to you and others in the future? We can use both uninformative and informative variable names

```
b <- 2/10
malaria_prevalence_2020 <- 2/10
```

The "<-" operator tells R to take the number to the right of the symbol and store it in the variable named on the left (You could also use"=")

Note: R is case sensitive so for example, the variable 'malaria_prevalence_2020' is different to 'Malaria_prevalence_2020'. Variable names cannot begin with numbers, and you cannot have 'spaces' in your variable names (we normally use an underscore '_' where we might want a space

You'll probably create lots of variable names, and you might forget some of them. You can list all the ones in your session using

```
ls()
```

Or by looking the in the 'environment' pane

You can overwrite a variable at any point, for example,

```
test_variable <- 15
test_variable
```

```
[1] 15
```

```
test_variable <- 20
test_variable
```

```
[1] 20
```

In R programming, the terms 'variable' and 'object' are often used interchangeably.

Note: some variable names are not allowed - variables cannot begin with a number or punctuation. It is not a good idea to name variables after in-built R functions - i.e not a good idea to name a variable 'sum' or 'mean' - we will see very shortly that these words already have a purpose in R!

Strings

Strings are like numbers (in that they represent only one object), but text instead:

```
my_first_string  <- "avocado"
my_first_string
```

```
[1] "avocado"
```

Now, try this:

```
my_first_string  * my_first_string
```

What happens? Why? If you're not sure what type of data your object is, you can use this function:

```
class(my_first_string)
```

```
[1] "character"
```

```
class(my_first_variable)
```

```
[1] "numeric"
```

Here, "character" is the name of the datatype of strings, and "double" is a type of number. You can also use the functions "typeof" and "str" to find out about an object (try them out!).

Vectors

Vectors and lists are the same kind of object, and represent a set of variables (numbers, strings) in an order. To do this, we use the function 'c' which is short for 'combine'. It combines various objects into a vector. Create these vectors in R:

```
v1 <- c(1,2,3,4,5)
v2 <- c(0.1,0.15,0.2,0.4,0.5)
v3 <- c("red","blue","green","orange","black")
```

You can access an individual element by knowing its position in the list. So the 3rd element in the list v1 is found by using square brackets:

```
v1[3]
```

```
[1] 3
```

Different types of brackets have different roles in R, so it's important you use the correct type.

- Square brackets - [] - are used to access elements of vectors (and dataframes and other complex structures)
- Round brackets - () - are used to contain the arguments of a function - i.e. in sqrt(25), the argument to the function 'sqrt' is '25'

> Question 1: What happens if you try to find an element that doesn't exist? (e.g. the 0th or 6th element of v1 - how do you type this and what is the output?)

You can calculate summary statistics of, and plot, a vector. What do each of these do?:

```
mean(v1)
sd(v1)
var(v1)
min(v1)
max(v1)
sum(v1)
sum(v1[c(1,4)])
length(v1)
plot(v1)
plot(v2,v1)
```

Question 2: What happens if you try to do use these operations on v3 rather than v1?

Question 3: Try v1* v2 - What has the operator * done to your vectors? Is that what you expected?

You can create an zero-vector of a given length (e.g. 14) like this:

```
v4 <- rep(0, 14)
```

This literally means - we want to repeat the value '0' 14 times – you can learn more about the 'rep' function by typing ?rep

You can then add values into your vector, for example:

```
v4[1] <- 10
v4
```

```
 [1] 10  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Question 4:What do you think the following will do?

`v4[2:14] <- c(11:23)` Try to guess before you try it!

Note: the colon operator ':' allows us to create vectors of consecutuve numbers. For example 1:5 creates a vector from 1 to 5. Quickly try creating a vector from 101 to 200.

Data frames

Data frames are tables that are used for storing data, similar to what you might be used to seeing in excel. R has lots of built-in data sets that you can practice on, including one called CO2 . Type:

```
?CO2
```

to read the documentation. You can load this dataset into R using:

```
data("CO2")
```

This will add a object call `CO2` into our environment, you should be able to see it in the Environment pane. To see the data, you could type CO2 but it's quite long. Instead you can look at the first few lines by typing

```
 head(CO2)
```

```
  Plant   Type  Treatment conc uptake
1   Qn1 Quebec nonchilled   95   16.0
2   Qn1 Quebec nonchilled  175   30.4
3   Qn1 Quebec nonchilled  250   34.8
4   Qn1 Quebec nonchilled  350   37.2
5   Qn1 Quebec nonchilled  500   35.3
6   Qn1 Quebec nonchilled  675   39.2
```

Or if you want to look at more lines (i.e. the first 20)

```
head(CO2, n = 20)
```

```
   Plant   Type  Treatment conc uptake
1    Qn1 Quebec nonchilled   95   16.0
2    Qn1 Quebec nonchilled  175   30.4
3    Qn1 Quebec nonchilled  250   34.8
4    Qn1 Quebec nonchilled  350   37.2
5    Qn1 Quebec nonchilled  500   35.3
6    Qn1 Quebec nonchilled  675   39.2
7    Qn1 Quebec nonchilled 1000   39.7
8    Qn2 Quebec nonchilled   95   13.6
9    Qn2 Quebec nonchilled  175   27.3
10   Qn2 Quebec nonchilled  250   37.1
11   Qn2 Quebec nonchilled  350   41.8
12   Qn2 Quebec nonchilled  500   40.6
13   Qn2 Quebec nonchilled  675   41.4
14   Qn2 Quebec nonchilled 1000   44.3
15   Qn3 Quebec nonchilled   95   16.2
16   Qn3 Quebec nonchilled  175   32.4
17   Qn3 Quebec nonchilled  250   40.3
18   Qn3 Quebec nonchilled  350   42.1
19   Qn3 Quebec nonchilled  500   42.9
20   Qn3 Quebec nonchilled  675   43.9
```

We can see how big this dataset is using the following command

```
dim(CO2)
```

```
[1] 84  5
```

Where the first number is the number of rows and the second number is the number of columns.

Q5: What are the column names of this dataset?

You can access individual element of a dataframe by knowing its row and column position. For example, "Quebec" is in third row and second column, so we can find it by typing:

```
CO2[3,2]
```

```
[1] Quebec
Levels: Quebec Mississippi
```

So the first number refers to the row and the second number the column - we can remember this as CO2[row, column]

You can also extract an individual column or row. To extract the sixth row:

```
CO2[6,]
```

```
  Plant   Type  Treatment conc uptake
6   Qn1 Quebec nonchilled  675   39.2
```

or third column:

```
CO2[,3]
```

```
 [1] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
 [7] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[13] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[19] nonchilled nonchilled nonchilled chilled    chilled    chilled
[25] chilled    chilled    chilled    chilled    chilled    chilled
[31] chilled    chilled    chilled    chilled    chilled    chilled
[37] chilled    chilled    chilled    chilled    chilled    chilled
[43] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[49] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[55] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[61] nonchilled nonchilled nonchilled chilled    chilled    chilled
[67] chilled    chilled    chilled    chilled    chilled    chilled
[73] chilled    chilled    chilled    chilled    chilled    chilled
[79] chilled    chilled    chilled    chilled    chilled    chilled
Levels: nonchilled chilled
```

You can even subset the data, for example if you wanted to create a new dataframe, CO2_op2, which contains all rows but only the second and third columns:

```
CO2_op2 <- CO2[,2:3]
head(CO2_op2)
```

```
    Type  Treatment
1 Quebec nonchilled
2 Quebec nonchilled
3 Quebec nonchilled
4 Quebec nonchilled
5 Quebec nonchilled
6 Quebec nonchilled
```

You can also check what types of data are in each column using the command

```
str(CO2)
```

```
Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame':  84 obs. of  5 va
 $ Plant    : Ord.factor w/ 12 levels "Qn1"<"Qn2"<"Qn3"<..: 1 1 1 1 1 1 1 2 2 2 ...
 $ Type     : Factor w/ 2 levels "Quebec","Mississippi": 1 1 1 1 1 1 1 1 1 1 ...
 $ Treatment: Factor w/ 2 levels "nonchilled","chilled": 1 1 1 1 1 1 1 1 1 1 ...
 $ conc     : num  95 175 250 350 500 675 1000 95 175 250 ...
 $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7 13.6 27.3 37.1 ...
 - attr(*, "formula")=Class 'formula'  language uptake ~ conc | Plant
  .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
 - attr(*, "outer")=Class 'formula'  language ~Treatment * Type
  .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
 - attr(*, "labels")=List of 2
  ..$ x: chr "Ambient carbon dioxide concentration"
  ..$ y: chr "CO2 uptake rate"
 - attr(*, "units")=List of 2
  ..$ x: chr "(uL/L)"
  ..$ y: chr "(umol/m^2 s)"
```

Three of the columns are called factors – these are often strings and correspond to a column on which you may want to analyse. They typically represent a variable that has a limited number of values – for example, sex, age group, or region would be considered as factors in a malaria dataset.

We can see a quick summary of the numeric variables using

```
summary(CO2)
```

```
    Plant             Type         Treatment       conc          uptake     
 Qn1    : 7   Quebec     :42   nonchilled:42   Min.   :  95   Min.   : 7.70  
 Qn2    : 7   Mississippi:42   chilled   :42   1st Qu.: 175   1st Qu.:17.90  
 Qn3    : 7                                    Median : 350   Median :28.30  
 Qc1    : 7                                    Mean   : 435   Mean   :27.21  
 Qc3    : 7                                    3rd Qu.: 675   3rd Qu.:37.12  
 Qc2    : 7                                    Max.   :1000   Max.   :45.50  
 (Other):42                                                                  
```

If we want to explore on column of a dataframe, we use the '$' operator – for example

```
CO2$Treatment
```

```
 [1] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
 [7] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[13] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[19] nonchilled nonchilled nonchilled chilled    chilled    chilled   
[25] chilled    chilled    chilled    chilled    chilled    chilled   
[31] chilled    chilled    chilled    chilled    chilled    chilled   
[37] chilled    chilled    chilled    chilled    chilled    chilled   
[43] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[49] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[55] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
[61] nonchilled nonchilled nonchilled chilled    chilled    chilled   
[67] chilled    chilled    chilled    chilled    chilled    chilled   
[73] chilled    chilled    chilled    chilled    chilled    chilled   
[79] chilled    chilled    chilled    chilled    chilled    chilled   
Levels: nonchilled chilled
```

Returns just the 'treatment' column

Another useful function to quickly explore data is 'table' – what does this follow command return?

```
table(CO2$Treatment)
```

```
nonchilled    chilled 
        42         42 
```

Q6a: What is the value in the 14th row and 5th column?

Q6b: What are the values in the 1st to 7th rows of the 4th column

Q6c: How many of the samples are from Quebec?

We can also use functions to summarise the numeric data

```
mean(CO2$uptake)
```

```
[1] 27.2131
```

Q7: What is the range and median of the uptake column?

What if we only want to know the mean of the 'uptake' from Quebec?

This is where we'll touch upon the fact that there are multiple ways to do almost *EVERYTHING* in R!

The traditional 'base R' approach uses the 'which' function to figure out which bits of the data we want to calculate our sum over –

```
which(CO2$Type == "Quebec")
```

This returns a vector with the positions in the column CO2$Type where the type is Quebec

This is then used inside square brackets to select only these elements of CO2$conc

```
CO2$conc[which(CO2$Type == "Quebec")]
```

```
mean(CO2$uptake[which(CO2$Type == "Quebec")])
```

```
[1] 33.54286
```

The **tidyverse** approach, which we are covering in depth tomorrow uses a function called *filter*

```
op1 = filter(CO2, Type == "Quebec")
mean(op1$uptake)
```

```
[1] 33.54286
```

We can also filter by 2 or more criteria

```
op2 = filter(CO2, Type == "Quebec", Treatment == "chilled")
mean(op2$uptake)
```

```
[1] 31.75238
```

Can you now fill in this table - we want the **MEDIAN** uptake value for each treatment-type combination:

Treatment	Type	
	Mississippi	Quebec
chilled		
nonchilled		

Packages

R packages are collections of functions and data sets developed by the community.

They increase the power of R by improving existing base R functionalities, or by adding new ones. There seems to be an R package that does almost everything – from the most widely used ones

- **tidyverse** - for data manipulation and analysis
- **sf** - everything spatial and map related
- **ggplot2** - for a wide range of plots

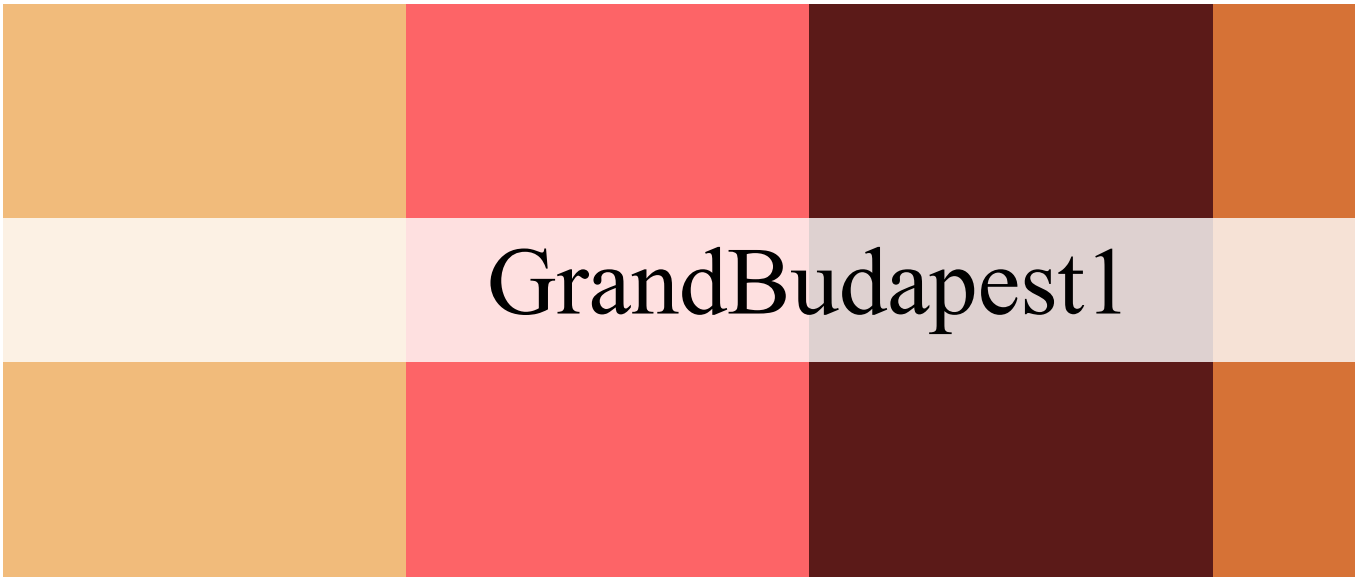
To the plain ridiculous:

- **wesanderson** - A library of colour pallettes based on Wes Anderson movies
- **cowsay** - Printed animals that say messages

```
names(wes_palettes)
```

```
 [1] "BottleRocket1"  "BottleRocket2"  "Rushmore1"      "Rushmore"
 [5] "Royal1"         "Royal2"         "Zissou1"        "Darjeeling1"
 [9] "Darjeeling2"    "Chevalier1"     "FantasticFox1"  "Moonrise1"
[13] "Moonrise2"      "Moonrise3"      "Cavalcanti1"    "GrandBudapest1"
[17] "GrandBudapest2" "IsleofDogs1"    "IsleofDogs2"
```

```
cols = wes_palette("GrandBudapest1")
cols
```




```
say("My favourite R package is called purrr", by = "cat")
```

```
 --------------
My favourite R package is called purrr
 --------------
    \
      \
        \
            |\___/|
          ==) ^Y^ (==
            \  ^  /
             )=*=(
            /     \
            |     |
           /| | | |\
           \| | |_|/\
      jgs  //_// ___/
```

_)

Installing and using a package

There are two stages to using a package

1) **Installing** – think of this as *buying a book and adding it to your library* – this is an action you only ever need to do once
2) **Loading** – think of this as *taking the book from your library shelf* – you need to do this each time you want to use it

We're going to try and install and use a package called **janitor** (https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html)

```
install.packages("janitor")
library(janitor)
```

When installing a new package, I typically either write it in the console or comment it out in my script to avoid reinstalling every time I run my code

```
# install.packages("janitor")
```

If you want to see all the functions available in a package, there are typically pdf's online of each function and often user guides
You can also remind yourself of functions in a package by typing the package name, followed by two colon marks, for example

```
janitor::
```

In your console or script

> Q8: Can you install and load either **wesanderson** or **cowsay** packages
>
> and generate either a Wes Anderson colour palette or a speaking animal?

Reading in data

Normally when we analyse data in R we are reading in external data sources - here we learn how to load those into R.

You have been sent a file called **training_case_data_wide.csv** - now manually save this into the **data** folder in your project directory

We can now read this into R

```
dat <- read_csv("data/training_case_data_wide.csv")
```

This is an aggregated dataset from DHIS2 in Ethiopia - we have a row for each woreda-month between January 2018 and December 2021 (Gregorian calendar).
In this dataset we have a date column (period), the region, zone, and woreda, and the number of presumed and confirmed malaria cases for each woreda month:

- presumed - number of unconfirmed malaria cases detected at the HF
- confirmed - number of RDT confirmed malaria cases detected at the HF

```
head(dat)
```

```
# A tibble: 6 x 7
  region zone     woreda     year period     confirmed presumed
  <chr>  <chr>    <chr>     <dbl> <date>         <dbl>    <dbl>
1 Somali Shabelle Aba-Korow  2018 2018-01-01        NA       NA
2 Somali Shabelle Aba-Korow  2018 2018-02-01        NA       NA
3 Somali Shabelle Aba-Korow  2018 2018-03-01        NA       NA
4 Somali Shabelle Aba-Korow  2018 2018-04-01        NA       NA
5 Somali Shabelle Aba-Korow  2018 2018-05-01        NA       NA
6 Somali Shabelle Aba-Korow  2018 2018-06-01        NA       NA
```

Note that (depending on your screen size) a few variables are not visible in the console. You can remedy this with the following code:

```
options(dplyr.width = Inf)
```

We can now apply some of the tools we've learnt today to analyse some aggregated DHIS2 data from Ethiopia.

Example: How many confirmed malaria cases there were in Ababo woreda in 2021?

```
op1 = filter(dat, woreda == "Ababo", year == 2021)
op1
```

```
# A tibble: 12 x 7
   region zone                 woreda  year period     confirmed presumed
   <chr>  <chr>                <chr>  <dbl> <date>         <dbl>    <dbl>
 1 Oromia Horo Gudru Wellega Ababo   2021 2021-01-01        79        3
 2 Oromia Horo Gudru Wellega Ababo   2021 2021-02-01        16       12
 3 Oromia Horo Gudru Wellega Ababo   2021 2021-03-01        13       NA
 4 Oromia Horo Gudru Wellega Ababo   2021 2021-04-01         2       NA
 5 Oromia Horo Gudru Wellega Ababo   2021 2021-05-01         5       NA
 6 Oromia Horo Gudru Wellega Ababo   2021 2021-06-01         6       NA
 7 Oromia Horo Gudru Wellega Ababo   2021 2021-07-01        14       NA
 8 Oromia Horo Gudru Wellega Ababo   2021 2021-08-01        22       NA
 9 Oromia Horo Gudru Wellega Ababo   2021 2021-09-01        56       NA
10 Oromia Horo Gudru Wellega Ababo   2021 2021-10-01        70        8
11 Oromia Horo Gudru Wellega Ababo   2021 2021-11-01        66       20
12 Oromia Horo Gudru Wellega Ababo   2021 2021-12-01        85        6
```

```r
sum(op1$confirmed)
```

```
[1] 434
```

> Question 9: What happens if we repeat this for presumed cases in Ababo in 2021?, Why?

It's really important to look at your data closely! We can fix this problem by using the following logic:

```r
test_vector <- c(1,5,8,3,NA,6)
sum(test_vector)
```

```
[1] NA
```

```r
sum(test_vector, na.rm=TRUE)
```

```
[1] 23
```

How did we know to do this? We looked at the help file for *sum* by using *?sum* and saw the option to add an argument that removes NAs

> Question 9 cont.: Now we know how to remove NAs, how many presumed malaria cases were reported in Ababo woreda in 2021?

We will spend tomorrow learning about more ways to manipulate and aggregate data - but for now, if you have time left, you can try and answer these questions using the examples above!

> Final exercise 1: What was the total number of cases (presumed and confirmed) in Hudet woreda in 2020?
>
> Final exercise 2: Were there more presumed or confirmed cases of malaria in Awra woreda in 2020?
>
> Final exercise 3: In 2021, were there more confirmed malaria cases in Hulet Ej Enese or Takusa?

Cheat sheet of functions we've learnt today

- sqrt()
- exp()
- log()
- rep()
- summary()
- table()
- dim()
- mean()
- median()
- range()
- sum()
- str()
- class()
- head()
- which()
- read.csv()

(And one functions from the tidyverse world, which will be the focus of tomorrow's session)

- filter()

1 Using `if` statements

Conditional statements are fundamental in programming, allowing for decision-making based on different conditions. In R, the primary conditional statement is the `if` statement, which can be used alone or combined with `else` and `else if` for more complex logic.

1.1 Basic If Statement

The basic `if` statement checks a condition and executes a block of code if the condition is true.

Syntax:

```
if (condition) {
  # code to execute if the condition is true
}
```

Example:

```
x <- 10

if (x > 5) {
  print("x is greater than 5")
}
```

1.2 If-Else Statement

To execute a block of code when the condition is false, use an `else` statement.

Syntax:

```
if (condition) {
  # code to execute if the condition is true
} else {
```

```
  # code to execute if the condition is false
}
```

Example:

```
x <- 3

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

1.2.1 If-Else If-Else Statement

For multiple conditions, use `else if` to check additional conditions.

Syntax:

```
if (condition1) {
  # code to execute if condition1 is true
} else if (condition2) {
  # code to execute if condition2 is true
} else {
  # code to execute if none of the conditions are true
}
```

Example:

```
x <- 7

if (x > 10) {
  print("x is greater than 10")
} else if (x > 5) {
  print("x is greater than 5 but less than or equal to 10")
} else {
  print("x is 5 or less")
}
```

1.3 Nested If Statements

You can also nest if statements inside one another to check multiple levels of conditions.

Example:

```
x <- 15
y <- 20

if (x > 10) {
  if (y > 15) {
    print("x is greater than 10 and y is greater than 15")
  } else {
    print("x is greater than 10 but y is not greater than 15")
  }
} else {
  print("x is 10 or less")
}
```

1.3.1 Vectorized If Statements

When working with vectors, `ifelse` is a more efficient way to apply conditional logic.

Syntax:

```
ifelse(test, yes, no)
```

- `test`: A logical condition.
- `yes`: The value to return if the condition is true.
- `no`: The value to return if the condition is false.

Example:

```
x <- c(2, 7, 5, 10)

result <- ifelse(x > 5, "Greater than 5", "5 or less")
print(result)
```

1.4 Conclusion

Using if statements in R allows you to execute code based on conditions, making your programs more dynamic and flexible. Whether you're using a simple `if` statement, adding `else` and `else if` for more complex logic, or applying conditional logic to vectors with `ifelse`, mastering these constructs is essential for effective programming in R.

2 More base R functions

This section contains additional functions and tools that we will use in later sections during the EpidemiaR Training workshop.

2.1 Concatenating strings and printing messages

In R, `paste()` and `paste0()` are functions used to concatenate strings, while `message()` is used to print messages to the console. Understanding these functions helps in creating more informative and readable outputs, as well as in debugging and providing user feedback.

2.1.1 `paste()` Function

The `paste()` function concatenates strings with a specified separator.

Syntax:

```
paste(..., sep = " ", collapse = NULL)
```

- `...`: One or more R objects to be concatenated.
- `sep`: A string to separate the terms (default is a space).
- `collapse`: An optional string to separate the results when concatenating vectors.

Examples:

1. Concatenating strings with spaces:

   ```
   str1 <- "Hello"
   str2 <- "World"
   result <- paste(str1, str2)
   print(result)
   # Output: "Hello World"
   ```

2. Using a different separator:

```r
result <- paste("A", "B", "C", sep = "-")
print(result)
# Output: "A-B-C"
```

3. Collapsing a vector into a single string:

```r
words <- c("apple", "banana", "cherry")
result <- paste(words, collapse = ", ")
print(result)
# Output: "apple, banana, cherry"
```

2.1.2 `paste0()` Function

The `paste0()` function is a shortcut for `paste(..., sep = "")`. It concatenates strings without any separator.

Syntax:

```r
paste0(...)
```

- `...`: One or more R objects to be concatenated.

Examples:

1. Concatenating strings without spaces:

```r
str1 <- "Hello"
str2 <- "World"
result <- paste0(str1, str2)
print(result)
# Output: "HelloWorld"
```

2. Concatenating multiple strings:

```r
result <- paste0("A", "B", "C")
print(result)
# Output: "ABC"
```

2.1.3 `message()` Function

The `message()` function is used to print a message to the console. Unlike `print()`, it sends its output to the standard message stream and is often used for warnings, informational messages, and debugging.

Syntax:

```
message(...)
```

- `...`: One or more R objects to be printed.

Examples:

1. Printing a simple message:

```
message("This is an informational message.")
# Output: This is an informational message.
```

2. Combining `paste()` with `message()`:

```
name <- "John"
age <- 30
message(paste("Name:", name, "Age:", age))
# Output: Name: John Age: 30
```

3. Combining `paste0()` with `message()`:

```
prefix <- "ID_"
id <- 1234
message(paste0("Generated ID: ", prefix, id))
# Output: Generated ID: ID_1234
```

2.2 Loading objects and data files

2.2.1 R objects and data

R provides several functions for saving and loading data, allowing you to preserve your workspace and share data with others. The `save()` and `load()` functions save and restore entire R workspaces, while `saveRDS()` and `readRDS()` handle individual R objects.

2.2.1.1 `save()` and `load()` Functions

The `save()` function saves R objects to a specified file, which can be loaded back into the R environment using the `load()` function.

save() Syntax:

```
save(..., file)
```

- `...`: R objects to be saved.
- `file`: A character string naming the file to save the data to.

load() Syntax:

```
load(file)
```

- `file`: A character string naming the file to load the data from.

Examples:

1. Saving multiple objects:

   ```
   x <- 1:10
   y <- letters[1:10]
   save(x, y, file = "data.RData")
   ```

2. Loading the saved objects:

   ```
   load("data.RData")
   print(x)
   print(y)
   ```

2.2.1.2 `saveRDS()` and `readRDS()` functions

The `saveRDS()` function saves a single R object to a file, and `readRDS()` restores it. Unlike `save()`, `saveRDS()` does not save the object name, so you can assign it any name when you load it.

saveRDS() Syntax:

```
saveRDS(object, file)
```

- `object`: The R object to be saved.

- `file`: A character string naming the file to save the object to.

readRDS() Syntax:

```
readRDS(file)
```

- `file`: A character string naming the file to read the object from.

Examples:

1. Saving a single object:

```
z <- matrix(1:9, nrow = 3)
saveRDS(z, file = "matrix.RDS")
```

2. Loading the saved object:

```
my_matrix <- readRDS("matrix.RDS")
print(my_matrix)
```

Using `load()`, `save()`, `saveRDS()`, and `readRDS()` functions in R enables you to efficiently save and restore your data. Whether you need to save entire workspaces or individual objects, these functions provide flexible options for data persistence and sharing.

2.2.2 CSV and Excel files

2.3 Loading CSV and Excel Files into R

Reading data from CSV and Excel files is a common task in data analysis. R provides functions to easily load these files into your workspace for analysis.

2.3.0.1 Loading CSV Files

To load CSV files, use the `read.csv()` or `readr::read_csv()` functions. The `readr` package's `read_csv()` is often preferred for its speed and efficiency.

`read.csv()` Syntax:

```
data <- read.csv(file, header = TRUE, sep = ",")
```

- `file`: Path to the CSV file.
- `header`: Logical value indicating if the file contains a header row.

- `sep`: Character separating the values (default is a comma).

Example:

```
data <- read.csv("path/to/yourfile.csv")
print(head(data))
```

`readr::read_csv()` Syntax:

```
library(readr)
data <- read_csv("path/to/yourfile.csv")
print(head(data))
```

2.3.0.2 Loading Excel Files

To load Excel files, use the `readxl` package, which provides the `read_excel()` function.

Syntax:

```
library(readxl)
data <- read_excel(path, sheet = 1)
```

- `path`: Path to the Excel file.
- `sheet`: Sheet number or name to read from (default is the first sheet).

Example:

```
library(readxl)
data <- read_excel("path/to/yourfile.xlsx", sheet = "Sheet1")
print(head(data))
```

Using `read.csv()`, `readr::read_csv()`, and `read_excel()` functions, you can easily load CSV and Excel files into R for analysis. These functions provide a straightforward way to import your data and begin your analysis quickly.

2.4 Source an entire R script

Sometimes you may want to run an entire script all at once. This can be useful when using multiple scripts that must be run in a structured order, or when storing all of your custom functions in a separate script.

We can use the `source()` function to execute an entire script at once.

Example:

```
script_file <- file.path("path/to/yourscript.R")
source(script_file)
```

This will run all of the operations in that script in your active R sessions. All libraries, variables, and functions that are loaded or created in your sourced script will then be available in your session.

You can use the `exists()` function to create to make sure that the path you provide leads to an existing script. This function will return either `TRUE` or `FALSE`. This is also useful for checking if a data file is present in an expected location before loading and execute additional functions, especially when combined in an `if` statement.

A few things to watch out for: 1. Loading libraries may cause name conflicts for functions with shared names. 2. Similarly, sourcing a script will overwrite existing variables with the same name. 3. Sourcing script may take a long time depending on the complexity of the script.

2.5 Additional base R functions

Here are some additional functions that you may encounter in during the EpidemiaR Training sections.

- `Sys.Date()`/`Sys.Time()` will return the active data or time.
- `nrow()` will return the number of rows in a dataframe.
- `length()` will return the number of objects in a vector, or the number of columns in a dataframe.
- `unique()` will return all of the unique entries in a vector.

3 Chaining multiple steps using pipes

> ⚠ Check your R version!
>
> Some of the training material will use a special operator called the "native pipe" (which looks like this: `|>`). This function was first included in R version 4.1, which was released in May 2021.
> If you haven't updated your R in while then it may be time to update. You can check what your current version is by running `R.version` in the console.

Pipes are a powerful tool in R that enable a more readable and efficient way of chaining multiple operations. The `%>%` pipe operator, introduced by the `magrittr` package and widely popularized by the `dplyr` package in the `tidyverse`, allows you to pass the result of one function directly into the next function.

3.1 Pipes in `tidyverse`

To use pipes, you'll need to install and load the `magrittr` or `tidyverse` package. Remember that you only need to install the package one time, but you will have to load `tidyverse` if you want to use the pipe in a given script.

Install `tidyverse`:

```
install.packages("tidyverse")
```

Load the `tidyverse` package:

```
library(tidyverse)
```

3.1.1 Basic Pipe Syntax

The pipe operator `%>%` takes the output from the left-hand side and uses it as the first argument for the function on the right-hand side.

Syntax:

```
data %>%
  function1() %>%
  function2() %>%
  function3()
```

3.1.1.1 Example without Pipes

Consider the following operations on a data frame without using pipes:

Example:

```
library(dplyr)

data <- mtcars
filtered_data <- filter(data, cyl == 6)
selected_data <- select(filtered_data, mpg, hp)
summarized_data <- summarize(selected_data, mean_mpg = mean(mpg), mean_hp = mean(hp))

print(summarized_data)
```

3.1.1.2 Example with Pipes

The same operations can be performed more concisely using pipes:

Example:

```
library(dplyr)

mtcars %>%
  filter(cyl == 6) %>%
  select(mpg, hp) %>%
  summarize(mean_mpg = mean(mpg), mean_hp = mean(hp)) %>%
  print()
```

3.1.2 Using Pipes with Custom Functions

Pipes can also be used with custom functions. Define your function and use it within a pipe.

Example:

```r
custom_function <- function(df) {
  df %>%
    filter(gear == 4) %>%
    select(mpg, wt)
}

mtcars %>%
  custom_function() %>%
  head()
```

3.1.3 Using the Dot Placeholder

Sometimes, the data does not automatically fit as the first argument in the next function. In such cases, use the dot (.) as a placeholder.

Example:

```r
mtcars %>%
  filter(cyl == 4) %>%
  select(mpg, wt) %>%
  {
    n <- nrow(.)
    mean_wt <- mean(.$wt)
    data.frame(n = n, mean_wt = mean_wt)
  }
```

3.1.4 Nesting Pipes

Pipes can be nested for more complex operations. This is particularly useful when combining multiple data frames or performing multi-step operations.

Example:

```r
data1 <- mtcars %>%
  filter(cyl == 6) %>%
  select(mpg, hp)

data2 <- mtcars %>%
  filter(cyl == 4) %>%
  select(mpg, wt)
```

```r
combined_data <- bind_rows(data1, data2)
print(combined_data)
```

Using pipes in R enhances code readability and efficiency, allowing you to write more concise and maintainable code. Whether performing simple data manipulations or complex data transformations, pipes streamline your workflow by reducing the need for intermediate variables and nested function calls.

3.2 Native Pipe Operator in R (R 4.1+)

Starting with R version 4.1, a native pipe operator (`|>`) was introduced, providing an alternative to the `%>%` pipe from the `magrittr` package. The native pipe is part of the base R language, eliminating the need to load external packages for basic piping operations.

3.2.1 Basic Syntax

The native pipe operator works similarly to the `%>%` operator but uses `|>` instead.

Syntax:

```r
data |>
  function1() |>
  function2() |>
  function3()
```

Here is an example using traditional function chaining without the native pipe:

Example:

```r
data <- mtcars
filtered_data <- filter(data, cyl == 6)
selected_data <- select(filtered_data, mpg, hp)
summarized_data <- summarize(selected_data, mean_mpg = mean(mpg), mean_hp = mean(hp))

print(summarized_data)
```

The same operations can be performed more concisely using the native pipe:

Example:

```r
mtcars |>
  (\(df) filter(df, cyl == 6))() |>
  (\(df) select(df, mpg, hp))() |>
  (\(df) summarize(df, mean_mpg = mean(mpg), mean_hp = mean(hp)))() |>
  print()
```

3.2.2 Using Native Pipe with Custom Functions

Native pipes can also be used with custom functions, providing a clean and readable way to chain operations.

Example:

```r
custom_function <- function(df) {
  df |>
    filter(gear == 4) |>
    select(mpg, wt)
}

mtcars |>
  custom_function() |>
  head()
```

3.2.3 Using the Dot Placeholder

While the native pipe does not use the dot (`.`) placeholder in the same way as `%>%`, it can still be used flexibly with anonymous functions.

Example:

```r
mtcars |>
  (\(df) filter(df, cyl == 4))() |>
  (\(df) select(df, mpg, wt))() |>
  (\(df) {
    n <- nrow(df)
    mean_wt <- mean(df$wt)
    data.frame(n = n, mean_wt = mean_wt)
  })()
```

The introduction of the native pipe operator in R 4.1 provides a built-in and efficient way to chain operations, enhancing code readability without relying on external packages. While it

lacks some of the tools of `%>%`, such as the dot placeholder, it integrates seamlessly with base R functions and custom workflows.

You may encounter both types of pipes, and most often they work interchangeably. However, it is important to note the stuble differences between the tidyverse and native pipes, especially when debugging errors.

Part II

Data manipulation with `dplyr` and `tidyr`

The power of packages

One of the great things about using R are the thousands of available packages, which provide additional functions for many analytical tasks, such as data cleaning, statistical modelling, mapping, and much more. R packages are open-source, which means that they are free to use and maintained by the R community.

Installing and loading packages

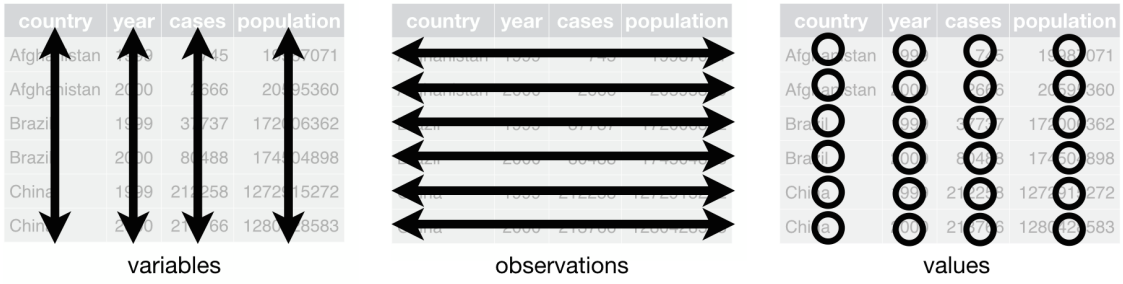
Throughout the rest of the workshop we will use a set of R packages manipulating data and creating plots and maps. As we covered in the Day 1 lesson, we first need to install the package on our computer using the `install.packages()` function. This only needs to be done one time (you probably already did this earlier).

```
install.packages("dplyr")
```

Once the package has been installed, we can load it into our current R session using the `library()` function. Unlike installing, you will need to load the library each time you want to use it. This is because some libraries may have functions with the same names as other libraries or as our variables.

```
library(dplyr)
```

For the next series of exercises, we will be using a group of packages which have been designed to work together to do common data science tasks. This group of packages is called the "Tidyverse", because it is designed to work within the "tidy" data philosophy:



Some important qualities of this philosophy is that our data should have the following format:

1. Each column should be a single variable with one data type.
2. Each row should be a single observation.

3. Each cell should be a single value contains **one** piece of information.

We can install all of these packages at once using `install.package("tidyverse")`. Remember that we only install the package once, so it is actually better to type this directly into the console instead of in our R script since it does not need to be repeated. Also be aware that this may take some time especially if internet quality is poor. After the package has finished installing it is ready to be loaded into our R session.

```
library(tidyverse)
```

Reading data into R

Once the **`tidyverse`** package is loaded into our session we will have access to all of the functions in each of the Tidyverse packages. This includes packages for loading, manipulating, and plotting data. The function we will use is `read_csv()` to read in the woreda-level data we worked with previously. Note that this is similar but slightly different to the read.csv() function we used in our previous exercise.

```
case_data <- read_csv("data/training_case_data_long.csv")
```

This is the same dataset we used in the Day 1 workshop, only this time we called the object `case_data` instead of `dat`. It's good practice to name your objects something short and meaningful, so that it's easy to type and remember (this is especially useful when you have multiple data objects).

Also, in this file the data are organized in "long" format, whereas the file used in Day 1 was in "wide" format. We will discuss the difference between "long" and "wide" formatted data in today's material, as well as how to change the shape of our data.

Inspecting data

Like in the previous exercise, we can use the `head()` and `summary()` functions to view aspects of the dataframe. We can also use the `view()` function to open the entire file in the R Studio Viewer, however view large files (millions of rows) can cause R Studio to crash. We can also install additional packages, such as **`skimr`** to get an even more detailed summary (run `install.package("skimr")` in your console).

```
head(case_data)
str(case_data)
summary(case_data)
```

```
# Using skimr package, remember to install first!
library(skimr)
skim(case_data)
```

Question 1: How many rows and columns are in case_data?

Question 2: What "type" of data are each column (character, vector, etc.)?

Data manipulation using `*tidyverse*`

In the Day 1 workshop we learned some built-in functions, or "base" functions, for simple data manipulations such as selecting a specific column or filter for only rows that match some criteria. In this lesson we will learn the `*tidyverse*` approach to these and additional common data manipulation tasks, using two packages called `*dplyr*` and `*tidyr*`. The `*dplyr*` package provides functions for the most common data manipulations jobs, and the `*tidyr*`package provides functions for reshaping or pivoting dataframes (similar to pivot tables in Microsoft Excel).

Selecting columns and filtering rows

To select a specific column from a dataframe, use the `select()` function. The first argument will always be the dataframe object that you're working with, followed by the name(s) of the column or columns you want to select.

```
# Select just one column (region)
select(case_data, region)
# Select multiple columns
select(case_data, woreda, data_type, count)
```

To select all the columns **except** certain ones, you can use a `-` in front of the column name.

```
# Select all but one column
select(case_data, -region)
# Removing multiple columns
select(case_data, -period, -region)
```

To choose specific rows based on some criteria, use `filter()`. Again, the first argument will be the dataframe, then the following argument will be the condition that we want use to subset the data.

```
  filter(case_data, region == "Oromia")
```

```
# A tibble: 32,064 x 7
   region zone               woreda  year period     data_type count
   <chr>  <chr>              <chr>  <dbl> <date>     <chr>     <dbl>
 1 Oromia Horo Gudru Wellega Ababo   2018 2018-01-01 confirmed    NA
 2 Oromia Horo Gudru Wellega Ababo   2018 2018-02-01 confirmed    NA
 3 Oromia Horo Gudru Wellega Ababo   2018 2018-03-01 confirmed    NA
 4 Oromia Horo Gudru Wellega Ababo   2018 2018-04-01 confirmed    NA
 5 Oromia Horo Gudru Wellega Ababo   2018 2018-05-01 confirmed    NA
 6 Oromia Horo Gudru Wellega Ababo   2018 2018-06-01 confirmed    NA
 7 Oromia Horo Gudru Wellega Ababo   2018 2018-07-01 confirmed    16
 8 Oromia Horo Gudru Wellega Ababo   2018 2018-08-01 confirmed    83
 9 Oromia Horo Gudru Wellega Ababo   2018 2018-09-01 confirmed    35
10 Oromia Horo Gudru Wellega Ababo   2018 2018-10-01 confirmed   167
# i 32,054 more rows
```

Notice here that just like in Day 1 use have to use a `==` sign for setting a condition. You can read this as saying, "choose the rows in `case_data` where region is equal to"Oromia". Also notice that the number of rows in the object has gone down from 94176 to 32064.

We can filter on multiple conditions at once using multiple arguments, using a `,` to state separate conditions.

```
  filter(case_data, region == "Oromia", data_type == "presumed")
```

```
# A tibble: 16,032 x 7
   region zone               woreda  year period     data_type count
   <chr>  <chr>              <chr>  <dbl> <date>     <chr>     <dbl>
 1 Oromia Horo Gudru Wellega Ababo   2018 2018-01-01 presumed     NA
 2 Oromia Horo Gudru Wellega Ababo   2018 2018-02-01 presumed     NA
 3 Oromia Horo Gudru Wellega Ababo   2018 2018-03-01 presumed     NA
 4 Oromia Horo Gudru Wellega Ababo   2018 2018-04-01 presumed     NA
 5 Oromia Horo Gudru Wellega Ababo   2018 2018-05-01 presumed     NA
 6 Oromia Horo Gudru Wellega Ababo   2018 2018-06-01 presumed     NA
 7 Oromia Horo Gudru Wellega Ababo   2018 2018-07-01 presumed     NA
 8 Oromia Horo Gudru Wellega Ababo   2018 2018-08-01 presumed     NA
 9 Oromia Horo Gudru Wellega Ababo   2018 2018-09-01 presumed     NA
10 Oromia Horo Gudru Wellega Ababo   2018 2018-10-01 presumed     NA
# i 16,022 more rows
```

By default, each of the conditions in `filter()` must be TRUE to remain in the subset, however there are special operators that allow for more complex conditional operations. The most common are the AND (`&`) and OR (`|`) operators. Here are some examples:

```
# region is Easter AND data type is presumed
filter(case_data, region == "Oromia" & data_type == "presumed")

# region is Oromia OR data type is presumed
filter(case_data, region == "Oromia" | data_type == "presumed")

# region is Afar OR Oromia, AND count is over 1,000
filter(case_data, region == "Afar" | region == "Oromia", count > 1000)
```

Question 3: Why did we not use & in the third example?

Another useful operator is the MATCH operator (`%in%`), which will return TRUE if a value matches any value in a list of possible options.

```
# Keep rows where data type could be presumed or confirmed
filter(case_data, data_type %in% c("presumed", "confirmed"))
```

```
# A tibble: 94,176 x 7
   region zone     woreda     year period     data_type count
   <chr>  <chr>    <chr>     <dbl> <date>     <chr>     <dbl>
 1 Somali Shabelle Aba-Korow  2018 2018-01-01 confirmed    NA
 2 Somali Shabelle Aba-Korow  2018 2018-02-01 confirmed    NA
 3 Somali Shabelle Aba-Korow  2018 2018-03-01 confirmed    NA
 4 Somali Shabelle Aba-Korow  2018 2018-04-01 confirmed    NA
 5 Somali Shabelle Aba-Korow  2018 2018-05-01 confirmed    NA
 6 Somali Shabelle Aba-Korow  2018 2018-06-01 confirmed    NA
 7 Somali Shabelle Aba-Korow  2018 2018-07-01 confirmed    21
 8 Somali Shabelle Aba-Korow  2018 2018-08-01 confirmed    13
 9 Somali Shabelle Aba-Korow  2018 2018-09-01 confirmed    37
10 Somali Shabelle Aba-Korow  2018 2018-10-01 confirmed    20
# i 94,166 more rows
```

```
# Keep rows from a group of selected woredas
study_woredas <- c("Adama Town", "Liban Jawi", "Mieso", "Wondo")
filter(case_data, woreda %in% study_woredas)
```

```
# A tibble: 384 x 7
```

```
   region zone       woreda      year period     data_type count
   <chr>  <chr>      <chr>      <dbl> <date>     <chr>     <dbl>
 1 Oromia East Shewa Adama Town  2018 2018-01-01 confirmed    NA
 2 Oromia East Shewa Adama Town  2018 2018-02-01 confirmed    NA
 3 Oromia East Shewa Adama Town  2018 2018-03-01 confirmed    NA
 4 Oromia East Shewa Adama Town  2018 2018-04-01 confirmed    NA
 5 Oromia East Shewa Adama Town  2018 2018-05-01 confirmed    NA
 6 Oromia East Shewa Adama Town  2018 2018-06-01 confirmed    NA
 7 Oromia East Shewa Adama Town  2018 2018-07-01 confirmed    NA
 8 Oromia East Shewa Adama Town  2018 2018-08-01 confirmed    NA
 9 Oromia East Shewa Adama Town  2018 2018-09-01 confirmed    NA
10 Oromia East Shewa Adama Town  2018 2018-10-01 confirmed    NA
# i 374 more rows
```

Question 4: Can you show all of the "presumed" data in the Somali region?

Question 5: Can you show all "confirmed" that have a count over 2000?

Finally, the `!` operator in R used for NOT or opposite conditions. The most common use cases are for using NOT EQUAL (`!=`) or does NOT MATCH operations.

```
# Keep all rows where region is NOT oromia
filter(case_data, region != "Addis Ababa")
```

```
# A tibble: 93,216 x 7
   region zone     woreda     year period     data_type count
   <chr>  <chr>    <chr>     <dbl> <date>     <chr>     <dbl>
 1 Somali Shabelle Aba-Korow  2018 2018-01-01 confirmed    NA
 2 Somali Shabelle Aba-Korow  2018 2018-02-01 confirmed    NA
 3 Somali Shabelle Aba-Korow  2018 2018-03-01 confirmed    NA
 4 Somali Shabelle Aba-Korow  2018 2018-04-01 confirmed    NA
 5 Somali Shabelle Aba-Korow  2018 2018-05-01 confirmed    NA
 6 Somali Shabelle Aba-Korow  2018 2018-06-01 confirmed    NA
 7 Somali Shabelle Aba-Korow  2018 2018-07-01 confirmed    21
 8 Somali Shabelle Aba-Korow  2018 2018-08-01 confirmed    13
 9 Somali Shabelle Aba-Korow  2018 2018-09-01 confirmed    37
10 Somali Shabelle Aba-Korow  2018 2018-10-01 confirmed    20
# i 93,206 more rows
```

Note that for NOT EQUAL the `!` operator comes next to the `=` sign, but for the NOT MATCH condition the `!` comes before the condition state. In the second case you can read that as, "do the opposite of this condition".

Question 6a: Create a table for all confirmed malaria cases in Amhara and Harari regions in 2020.

Question 6b: Create a table for all presumed malaria cases NOT in Amhara and Harari regions in 2020.

Question 7a: Create a table for all presumed and confirmed cases that are over 500.

Question 7b: Create a table for all data that are NOT presumed and that are over 500.

The types of conditional states that you can use depends on the type of column you want to base your `filter()`on. For example, `filter(case_data, count > 1000)` makes sense since the `count` column contains `numeric` data. However, `filter(case_data, region > 1000)` doesn't make sense since the `region` column contains `character` data. The rule of thumb is that the value you use to set your condition should match the "type" of data in selected column.

In the next section, we see how to deal with a special case:

Working with dates using the *lubridate* package

In the "tidy" data approach to working with data each column is a specific type of data, each row is an observation, and each cell is an individual value which conveys a single piece of information. Our dataset matches this philosophy, except for the "period" values, which contain information on the year, month, and day of the observation.

We could create separate columns for the year, month, and day, but this may complicate our filtering. For instance, what happens if we want to filter for a study period that continues across over parts of adjacent months or year? Such a common task would require complex set of conditional statements to filter correctly.

The *lubridate* package provides a number of functions to make working with data much easier. This is not included in *tidyverse*, so we have to install and then load it into our session.

```
# install.packages(lubridate)
library(lubridate)
```

The `ymd()` function allows us to create a `Date` class object based on the string input for YEAR-MONTH-DAY:

```r
# Vector of workshop days
workshop_days <- c("2023-09-04", "2023-09-05", "2023-09-06", "2023-09-07","2023-09-08")
class(workshop_days)
```

```
[1] "character"
```

```r
# Convert to a Date class
workshop_days <- ymd(workshop_days)
class(workshop_days)
```

```
[1] "Date"
```

Once you have a Date class object, `*lubridate*` provides many, many functions for working with date information. The primary functions we will use in this workshop are `year()` and `month()`, but there are many more in this `*lubridate*` cheatsheet.

```r
year(workshop_days)
```

```
[1] 2023 2023 2023 2023 2023
```

```r
month(workshop_days)
```

```
[1] 9 9 9 9 9
```

```r
month(workshop_days, label = TRUE)
```

```
[1] Sep Sep Sep Sep Sep
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

These functions can be used in `filter()`.

```r
filter(case_data, year(period) == 2020)
```

```
# A tibble: 23,544 x 7
   region zone     woreda     year period     data_type count
   <chr>  <chr>    <chr>     <dbl> <date>     <chr>     <dbl>
 1 Somali Shabelle Aba-Korow  2020 2020-01-01 confirmed    NA
 2 Somali Shabelle Aba-Korow  2020 2020-02-01 confirmed    NA
 3 Somali Shabelle Aba-Korow  2020 2020-03-01 confirmed    NA
 4 Somali Shabelle Aba-Korow  2020 2020-04-01 confirmed    NA
 5 Somali Shabelle Aba-Korow  2020 2020-05-01 confirmed    NA
 6 Somali Shabelle Aba-Korow  2020 2020-06-01 confirmed    NA
 7 Somali Shabelle Aba-Korow  2020 2020-07-01 confirmed    NA
 8 Somali Shabelle Aba-Korow  2020 2020-08-01 confirmed    NA
 9 Somali Shabelle Aba-Korow  2020 2020-09-01 confirmed    15
10 Somali Shabelle Aba-Korow  2020 2020-10-01 confirmed    NA
# i 23,534 more rows
```

```
  filter(case_data, between(period, ymd("2019-01-01"), ymd("2019-06-30")))
```

```
# A tibble: 11,772 x 7
   region zone                 woreda     year period     data_type count
   <chr>  <chr>                <chr>     <dbl> <date>     <chr>     <dbl>
 1 Somali Shabelle             Aba-Korow  2019 2019-01-01 confirmed    NA
 2 Somali Shabelle             Aba-Korow  2019 2019-02-01 confirmed    NA
 3 Somali Shabelle             Aba-Korow  2019 2019-03-01 confirmed    NA
 4 Somali Shabelle             Aba-Korow  2019 2019-04-01 confirmed    NA
 5 Somali Shabelle             Aba-Korow  2019 2019-05-01 confirmed    NA
 6 Somali Shabelle             Aba-Korow  2019 2019-06-01 confirmed    NA
 7 Afar   Zone 2 (Kilbet Rasu) Aba 'Ala   2019 2019-01-01 confirmed    55
 8 Afar   Zone 2 (Kilbet Rasu) Aba 'Ala   2019 2019-02-01 confirmed    26
 9 Afar   Zone 2 (Kilbet Rasu) Aba 'Ala   2019 2019-03-01 confirmed     9
10 Afar   Zone 2 (Kilbet Rasu) Aba 'Ala   2019 2019-04-01 confirmed    25
# i 11,762 more rows
```

Question 8: What were the reported total cases (presumed and confirmed) in Bambasi woreda each month during 2018?

Question 9: How many cases (presumed and confirmed) were reported in April 2020 in Gudetu Kondole woreda?

Question 10 (HARD): What were the monthly confirmed cases in Bambasi during the peak transmission season (~September-December) each year?

Creating new columns with `mutate()`

Another common task is creating new columns based on values in existing columns. The `*dplyr*` function for this action is `mutate()`.

Here is an example using the `*lubridate*` function from the section above to make a column for the month of observation:

```
mutate(case_data, month = month(period))
```

```
# A tibble: 94,176 x 8
   region zone     woreda     year period     data_type count month
   <chr>  <chr>    <chr>     <dbl> <date>     <chr>     <dbl> <dbl>
 1 Somali Shabelle Aba-Korow  2018 2018-01-01 confirmed    NA     1
 2 Somali Shabelle Aba-Korow  2018 2018-02-01 confirmed    NA     2
 3 Somali Shabelle Aba-Korow  2018 2018-03-01 confirmed    NA     3
 4 Somali Shabelle Aba-Korow  2018 2018-04-01 confirmed    NA     4
 5 Somali Shabelle Aba-Korow  2018 2018-05-01 confirmed    NA     5
 6 Somali Shabelle Aba-Korow  2018 2018-06-01 confirmed    NA     6
 7 Somali Shabelle Aba-Korow  2018 2018-07-01 confirmed    21     7
 8 Somali Shabelle Aba-Korow  2018 2018-08-01 confirmed    13     8
 9 Somali Shabelle Aba-Korow  2018 2018-09-01 confirmed    37     9
10 Somali Shabelle Aba-Korow  2018 2018-10-01 confirmed    20    10
# i 94,166 more rows
```

First, state the name for the new column, then `=` followed by the function for the new value. You can create multiple new columns in a single `mutate()` call, using a `,` to separate each column.

```
mutate(case_data,
    month_num = month(period),
    month_name = month(period, label = TRUE))
```

```
# A tibble: 94,176 x 9
   region zone     woreda   year period     data_type count month_num month_name
   <chr>  <chr>    <chr>   <dbl> <date>     <chr>     <dbl>     <dbl> <ord>
 1 Somali Shabelle Aba-Ko~  2018 2018-01-01 confirmed    NA         1 Jan
 2 Somali Shabelle Aba-Ko~  2018 2018-02-01 confirmed    NA         2 Feb
 3 Somali Shabelle Aba-Ko~  2018 2018-03-01 confirmed    NA         3 Mar
 4 Somali Shabelle Aba-Ko~  2018 2018-04-01 confirmed    NA         4 Apr
 5 Somali Shabelle Aba-Ko~  2018 2018-05-01 confirmed    NA         5 May
```

```
 6 Somali Shabelle Aba-Ko~  2018 2018-06-01 confirmed    NA         6 Jun
 7 Somali Shabelle Aba-Ko~  2018 2018-07-01 confirmed    21         7 Jul
 8 Somali Shabelle Aba-Ko~  2018 2018-08-01 confirmed    13         8 Aug
 9 Somali Shabelle Aba-Ko~  2018 2018-09-01 confirmed    37         9 Sep
10 Somali Shabelle Aba-Ko~  2018 2018-10-01 confirmed    20        10 Oct
# i 94,166 more rows
```

Remember that if you want to save any changes you will have to save the output into an object using the <- assignment operator. Otherwise then will not be updated in your variable.

```
case_data_dates <- mutate(case_data,
    month_num = month(period),
    month_name = month(period, label = TRUE))
```

In later sections we will see how to use `mutate()` to make calculations.

> Question 11: Can you add a new variable (column) to the dataset that gives the **quarter** of the year?
>
> Question 12: Can you add a new variable (column) to the dataset that gives just the last 2 digits of the year? i.e. 2021 becomes 21

Use Pipes to combine steps

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

For intermediate steps, we need to create a new intermediate object for the output of our first function, which will then be used as an input for the second function:

```
case_data_oromia_confirmed <- filter(case_data, region == "Oromia", data_type == "confirme
case_data_oromia_woreda_months <- select(case_data_oromia_confirmed, woreda, period, count
case_data_oromia_woreda_months
```

```
# A tibble: 16,032 x 3
   woreda period     count
   <chr>  <date>     <dbl>
 1 Ababo  2018-01-01    NA
 2 Ababo  2018-02-01    NA
 3 Ababo  2018-03-01    NA
 4 Ababo  2018-04-01    NA
```

```
 5 Ababo  2018-05-01    NA
 6 Ababo  2018-06-01    NA
 7 Ababo  2018-07-01    16
 8 Ababo  2018-08-01    83
 9 Ababo  2018-09-01    35
10 Ababo  2018-10-01   167
# i 16,022 more rows
```

This approach is readable, but it can quickly clutter up your workspace and take up additional memory. And if you're trying to use meaningful object names it can get tedious quickly.

You can also nest the functions (one function inside of the another).

```
case_data_oromia_woreda_months <- select(
  filter(case_data, region == "Oromia", data_type == "confirmed"),
  woreda, period, count)
```

This doesn't clutter the workshop or take up unnecessary memory, but it is difficult to read especially since R will interpret these steps from the inside out (first filter, then select).

The last option is to use pipes, a new addition to R. A pipe lets you take the output from one function and input it directly into the next function. By default, this will automatically go into the first argument of the new function. This is useful for stringing together multiple data cleaning steps while maintaining readability and keeping our environment clear. The *tidyverse* package includes a pipe function which looks like %>%. In RStudio, the shortcut for this pipe is Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac. You can adjust this shortcut under Tools » Modify Keyboard Shortcuts…

Here's an example of using a pipe for combine the filter and select from the previous example.

```
case_data %>%
  filter(region == "Oromia", data_type == "confirmed") %>%
  select(woreda, period, count)
```

```
# A tibble: 16,032 x 3
   woreda period     count
   <chr>  <date>     <dbl>
 1 Ababo  2018-01-01    NA
 2 Ababo  2018-02-01    NA
 3 Ababo  2018-03-01    NA
 4 Ababo  2018-04-01    NA
 5 Ababo  2018-05-01    NA
 6 Ababo  2018-06-01    NA
```

```
 7 Ababo  2018-07-01    16
 8 Ababo  2018-08-01    83
 9 Ababo  2018-09-01    35
10 Ababo  2018-10-01   167
# i 16,022 more rows
```

In this code, we used a pipe to send `case_data` into a `filter()` function and keep the rows for confirmed cases in Oromia region, then used another pipe to send that output into a `select()` where we only kept the woreda, period, and count columns. We didn't need to explicitly state the data object in the filter and select because data is always the first argument.

You may find it helpful to read the pipe like the word "then". Take the case, then filter for Oromia region and confirmed cases, then select the woredas, periods, and counts. We can also save this into a new object.

```
oromia_confirmed_cases <- case_data %>%
  filter(region == "Oromia", data_type == "confirmed") %>%
  select(woreda, period, count)
```

> Question 13: Using pipes, create a table contain the confirmed cases in January 2019 for each woreda in Oromia region. The table should only have two columns (woreda and count)

Grouping and summarizing data

Another common data manipulation task involves grouping data together and applying summary functions such as calculating means or totals. We can do some of these types of operations already. For instance, we can get the total number of presumed cases in Oromia region.

```
oromia_presumed <- case_data %>%
  filter(data_type == "presumed", region == "Oromia")
sum(oromia_presumed$count, na.rm = T)
```

```
[1] 58132
```

But what if we want to get summaries for each region at once? We could repeat the steps above, separating each region, calculating the totals, and then grouping these summaries back together. In programming this concept is often referred to as the **split-apply-combine** paradigm. The key `*dplyr*` functions for these tasks are `group_by()` and `summarize()` (you can also use the "proper" `summarise()` spelling as well).

First, `group_by()` takes in a column that contains categorical data, then use `summarize()` to calculate new summary statistics.

```
case_data %>%
  filter(data_type == "presumed") %>%
  group_by(region) %>%
  summarise(mean_presumed = mean(count, na.rm = TRUE))
```

```
# A tibble: 11 x 2
   region           mean_presumed
   <chr>                    <dbl>
 1 Addis Ababa              13.2
 2 Afar                     52.2
 3 Amhara                   24.9
 4 Benishangul Gumz         45.5
 5 Dire Dawa                 6.54
 6 Gambela                  70.3
 7 Harari                   12.2
 8 Oromia                   16.9
 9 SNNP                     29.6
10 Somali                   54.8
11 Tigray                   56.6
```

You can also group by more than one column, and output multiple columns within a single `summarize()` call.

```
case_data %>%
  filter(data_type == "presumed") %>%
  group_by(region, woreda, year) %>%
  summarise(mean_presumed_per_month = mean(count, na.rm = TRUE),
            total_presumed_per_month = sum(count, na.rm = TRUE))
```

```
# A tibble: 3,924 x 5
# Groups:   region, woreda [981]
   region      woreda         year mean_presumed_per_mo~1 total_presumed_per_m~2
   <chr>       <chr>         <dbl>                  <dbl>                  <dbl>
 1 Addis Ababa Addis Ketema   2018                   19.3                    212
 2 Addis Ababa Addis Ketema   2019                   66.6                    666
 3 Addis Ababa Addis Ketema   2020                   15.8                    158
 4 Addis Ababa Addis Ketema   2021                    2.67                    24
 5 Addis Ababa Akaki - Kalit  2018                   24.7                    272
```

```
 6 Addis Ababa Akaki - Kalit  2019                  66.3                      729
 7 Addis Ababa Akaki - Kalit  2020                  30.1                      331
 8 Addis Ababa Akaki - Kalit  2021                  27.8                      278
 9 Addis Ababa Arada          2018                   1.75                      14
10 Addis Ababa Arada          2019                   2.14                      15
# i 3,914 more rows
# i abbreviated names: 1: mean_presumed_per_month, 2: total_presumed_per_month
```

```
case_data %>%
  filter(data_type == "presumed") %>%
  group_by(region, woreda, year) %>%
  summarise(mean_presumed_per_month = mean(count, na.rm = TRUE),
            total_presumed_per_month = sum(count, na.rm = TRUE))
```

```
# A tibble: 3,924 x 5
# Groups:   region, woreda [981]
   region      woreda         year mean_presumed_per_mo~1 total_presumed_per_m~2
   <chr>       <chr>         <dbl>                  <dbl>                   <dbl>
 1 Addis Ababa Addis Ketema   2018                  19.3                      212
 2 Addis Ababa Addis Ketema   2019                  66.6                      666
 3 Addis Ababa Addis Ketema   2020                  15.8                      158
 4 Addis Ababa Addis Ketema   2021                   2.67                      24
 5 Addis Ababa Akaki - Kalit  2018                  24.7                      272
 6 Addis Ababa Akaki - Kalit  2019                  66.3                      729
 7 Addis Ababa Akaki - Kalit  2020                  30.1                      331
 8 Addis Ababa Akaki - Kalit  2021                  27.8                      278
 9 Addis Ababa Arada          2018                   1.75                      14
10 Addis Ababa Arada          2019                   2.14                      15
# i 3,914 more rows
# i abbreviated names: 1: mean_presumed_per_month, 2: total_presumed_per_month
```

Sometimes it is useful to rearrange the result of our summarized dataset, in which case we can use the `arrange()` function.

```
case_data %>%
  filter(data_type == "presumed",
         year(period) == 2019) %>%
  group_by(woreda, period) %>%
  summarise(total_presumed_per_month = sum(count)) %>%
  arrange(total_presumed_per_month)
```

```
# A tibble: 11,760 x 3
# Groups:   woreda [980]
   woreda       period     total_presumed_per_month
   <chr>        <date>                         <dbl>
 1 Ada'a        2019-10-01                         1
 2 Ada'a        2019-11-01                         1
 3 Addi Arekay  2019-09-01                         1
 4 Addis Ketema 2019-09-01                         1
 5 Adhaki       2019-10-01                         1
 6 Adigrat Town 2019-01-01                         1
 7 Adola Town   2019-04-01                         1
 8 Adola Town   2019-08-01                         1
 9 Adola Town   2019-09-01                         1
10 Adola Town   2019-11-01                         1
# i 11,750 more rows
```

By default arranging with be in ascending order, you can use `desc()` to make the output descending.

```
case_data %>%
  filter(data_type == "presumed",
         year(period) == 2019) %>%
  group_by(woreda, period) %>%
  summarise(total_presumed_per_month = sum(count)) %>%
  arrange(desc(total_presumed_per_month))
```

```
# A tibble: 11,760 x 3
# Groups:   woreda [980]
   woreda           period     total_presumed_per_month
   <chr>            <date>                        <dbl>
 1 Goro Baqaqsa     2019-09-01                      838
 2 Tselemti         2019-11-01                      801
 3 Tsegede (Tigray) 2019-10-01                      628
 4 Tsegede (Tigray) 2019-11-01                      620
 5 Mirab Armacho    2019-10-01                      616
 6 Metema           2019-10-01                      606
 7 Tselemti         2019-12-01                      575
 8 Jawi             2019-10-01                      557
 9 Chwaka           2019-11-01                      551
10 Bambasi          2019-10-01                      550
# i 11,750 more rows
```

Question 14: What were the total number of confirmed cases in each region in 2019?

Question 15: What were the total number of cases (presumed and confirmed) in each month for all regions for each year?

Question 16 (HARD): What were the total number of cases (presumed and confirmed) in the peak (September - December) and the low (January - April) transmission seasons for each woreda in Somali region during 2020?

Reshaping data with *tidyr*

So far we have covered a bunch of *dplyr* functions for manipulating data, most of which have changed the number of rows and/or columns in our dataframe. However, even though the columns, rows, and values have changed none of these have changed the "structure" of the dataframe. At the end of each function or piped function, the output always followed the conditions we discussed early:

1. Each column should be a single variable with one data type.
2. Each row should be a single observation.
3. Each cell should be a single value contains **one** piece of information.

This is commonly referred to as "long" format data, and often this means that there are relatively more rows than columns. Typically it is best to work in "long" format data, especially in R, however there are instances when we may want to change the "shape" of our data into the "wide" format. In Microsoft Excel this would be called creating a Pivot Table.

The *tidyr* package provides functions for reshaping data, including creating "wide" format pivot tables. To illustrate, lets take a look at records for a single woreda at a single timepoint.

```
case_data %>%
  filter(woreda == "Awabel", period == ymd("2020-01-01"))
```

```
# A tibble: 2 x 7
  region zone       woreda  year period     data_type count
  <chr>  <chr>      <chr>  <dbl> <date>     <chr>     <dbl>
1 Amhara East Gojam Awabel  2020 2020-01-01 confirmed    55
2 Amhara East Gojam Awabel  2020 2020-01-01 presumed      2
```

The resulting table has six rows, because there are six different types of records included. But what if we wanted to create a table where there is a separate column for each type of record? The `pivot_wider()` function will allow us to create this kind of "wide" format table. This

function requires us to state the column which our column names will come from (`names_from`), and which column the values in the new columns will come from (`values_from`).

```
case_data %>%
  filter(woreda == "Awabel", period == ymd("2020-01-01")) %>%
  pivot_wider(names_from = data_type, values_from = count)
```

```
# A tibble: 1 x 7
  region zone       woreda  year period     confirmed presumed
  <chr>  <chr>      <chr>  <dbl> <date>         <dbl>    <dbl>
1 Amhara East Gojam Awabel  2020 2020-01-01        55        2
```

The resulting output just has one row, but new columns for each of the types of records. This "wide" format is often useful for creating summary tables.

The opposite function is called `pivot_longer()`, which will take in a "wide" format table and output a "long" format table. For `pivot_longer()` we need to state the column name for the "key" which provides the label, the column names for the values, and which columns we want to pivot on. Here is how we can convert the above example from "wide" format to "long" format.

```
wide_data <- case_data %>%
  filter(woreda == "Awabel", period == ymd("2020-01-01")) %>%
  pivot_wider(names_from = data_type, values_from = count)


wide_data %>%
  pivot_longer(
    names_to = "data_type", values_to = "count",
    cols = c(confirmed, presumed))
```

```
# A tibble: 2 x 7
  region zone       woreda  year period     data_type count
  <chr>  <chr>      <chr>  <dbl> <date>     <chr>     <dbl>
1 Amhara East Gojam Awabel  2020 2020-01-01 confirmed    55
2 Amhara East Gojam Awabel  2020 2020-01-01 presumed      2
```

> Question 17: Create a table that contains the total number of confirmed cases each year for each region, then pivot wider to make it so the rows are the year and there is a column for each region.

Think of this as a multi-step process
1. Filter for confirmed cases
2. Work out which columns you are grouping by (i.e. what are we grouping over….think geography/time?)
3. Look at the output so far - what do we now want to sum over for our table?
4. Now we need to pivot wider - we want to make new columns - where do those names to come from (names_from)?, where will we find the values to populate the new cells (values_from)?

> Question 17b: Can you make this table longer again, where we now have 3 columns - year, region and count

> Question 18: Create a table that contains the total number of confirmed cases for each woreda and each year in Somali region, now pivot wider to make each woreda its own column where year is now the row - start by writing out the steps 1 by 1 as above

Final exercises

(1) For each region, calculate the total number of cases confirmed by RDT each year and present as a wide dataframe, where each column is a year and each row is a region and finally, order the dataframe such that the region with the most cases in 2020 is at the top

(2) For each woreda in Oromia region, calculate the proportion of total cases in 2020 that were confirmed (confirmed cases / confirmed cases + presumed)

Tips:

1. Start by filtering your data - what geographical region, year, and data types do we need?

2. Now we need to pivot wider - how can we now have data_type as our column names?

3. Now we need to group_by - what are we grouping by?

4. Almost there!! Now we need to sum up some columns - how can we do this?

5. Now to calculate the proportion - let's use the mutate function

6. Now let's use the 'select' function to retrieve the columns we want

Function cheatsheet

- Main cheat sheet page
- `*dplyr*` cheat sheet link
- `lubridate` cheat sheet
- `tidyr` cheat sheet

`*dplyr*` functions

- `select()`: subset columns
- `filter()`: subset rows on condition
- `mutate()`: create new columns
- `group_by()`: group data by one or more column
- `summarise()`: create summaries from dataframe (works within groups)
- `arrange()`: reorder dataframe based on ascending order (use `desc()` to invert)

`*tidyr*` functions

- `pivot_wider()`: go from "long" to "wide" format
- `pivot_longer()`: go from "wide" to "long" format

`*lubridate*` functions

- `ymd()`: convert class to date object based on "YYYY-MM-DD"
- `year()`: return the year from a date input
- `month()`: return the month from a date input
- `quarter()`: return the year quarter (1,2,3,4) from a date input

4 Joining data in tidyverse

Joining dataframes is a common task in data analysis, enabling you to combine datasets based on common keys. The `dplyr` package in the `tidyverse` provides several functions for different types of joins. This tutorial will focus on the `left_join()` function.

4.1 Joins overview

4.1.1 Installing and Loading the `tidyverse` Package

First, ensure that you have the `tidyverse` package installed and loaded.

Load the `tidyverse` package:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.4.3     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.0
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

4.1.2 Example Dataframes

Let's create two example dataframes for illustration.

Creating Dataframe 1:

```r
df1 <- tibble(
  id = c(1, 2, 3, 4),
  name = c("Alice", "Bob", "Charlie", "David")
)
print(df1)
```

```
# A tibble: 4 x 2
     id name
  <dbl> <chr>
1     1 Alice
2     2 Bob
3     3 Charlie
4     4 David
```

Creating Dataframe 2:

```r
df2 <- tibble(
  id = c(1, 2, 4, 5),
  score = c(85, 90, 88, 92)
)
print(df2)
```

```
# A tibble: 4 x 2
     id score
  <dbl> <dbl>
1     1    85
2     2    90
3     4    88
4     5    92
```

4.1.3 Using `left_join()`

The `left_join()` function combines rows from `df1` with matching rows from `df2`. If there is no match, the result will contain `NA` for the columns from `df2`.

Syntax:

```r
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)
```

- `x`: The left dataframe.

- `y`: The right dataframe.
- `by`: A character vector of variables to join by. If `NULL`, defaults to all variables with common names in `x` and `y`.

Example:

```
joined_df <- left_join(df1, df2, by = "id")
print(joined_df)
```

```
# A tibble: 4 x 3
     id name    score
  <dbl> <chr>   <dbl>
1     1 Alice      85
2     2 Bob        90
3     3 Charlie    NA
4     4 David      88
```

4.1.4 Handling Different Column Names

If the key columns have different names in the dataframes, use the `by` argument to specify the columns to join by.

Example with Different Column Names:

```
df3 <- tibble(
  student_id = c(1, 2, 4, 5),
  grade = c("A", "B", "B+", "A-")
)

joined_df2 <- left_join(df1, df3, by = c("id" = "student_id"))
print(joined_df2)
```

```
# A tibble: 4 x 3
     id name    grade
  <dbl> <chr>   <chr>
1     1 Alice   A
2     2 Bob     B
3     3 Charlie <NA>
4     4 David   B+
```

4.2 Example from training data

Comparing cases between different woredas is sometimes desirable when we compare 'apples with apples', and make the variable normalized for the population in each woreda. Let us look at the following question.

> Question 1: We want to identify woredas that have the highest annual confirmed cases per population in 2020.

To do this task, we need two datasets: one with confirmed cases, and another with population totals, both at the woreda-level. First, let's load the two datasets we will be using.

```
confirmed_cases_annual<- read_csv("data/training_case_data_long.csv")
```

```
Rows: 94176 Columns: 7
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (4): region, zone, woreda, data_type
dbl  (2): year, count
date (1): period

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
population_annual <- read_csv("data/training_population_data_long.csv")
```

```
Rows: 4905 Columns: 5
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): region, zone, woreda
dbl (2): year, population

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Before we join two data frames, we need to identify which columns we can use to join them together.

There are two important things we need to do before we can do the join.

- Identify which columns are common in both data frames. Note that the columns we use for joining may not have similar names in both data frames.
- Make sure the common columns (fields) have the same data type. We cannot use columns that have different data types to join two data frames.

The following script does the task of joining the two data frames, where we choose the case data to be the receiving data frame and the population dataset the server data frame.

```
annual_incidence <- confirmed_cases_annual %>%
  filter( year == 2020) %>%
  group_by (region, zone, woreda, year) %>%
  summarise(annual_cases = sum(count, na.rm=TRUE)) %>%
  left_join(population_annual, by = c("region" = "region",
                                      "zone" = "zone",
                                      "woreda" = "woreda",
                                      "year" = "year")) %>%
  mutate(api = annual_cases/ population * 1000) %>%
  arrange(desc(api))
```

```
`summarise()` has grouped output by 'region', 'zone', 'woreda'. You can
override using the `.groups` argument.
```

The function `left_join()` assumes that the data frame on the left of it is the master (receiver) while the one on its right side is the server data frame. This makes sure all rows on the left side will be included in the output even if there are no corresponding data rows in the data frame on the right side of the function. Any observations from the left side with no match on the right will have an `NA` value associated with them in the new joined dataset for the joined fields.

4.3 Conclusion

Using the `left_join()` function from the `dplyr` package in the `tidyverse`, you can easily join dataframes based on common keys. This is useful for combining related datasets and performing comprehensive data analysis. By understanding the basics of joins, you can leverage the power of `dplyr` to handle more complex data manipulations.