



























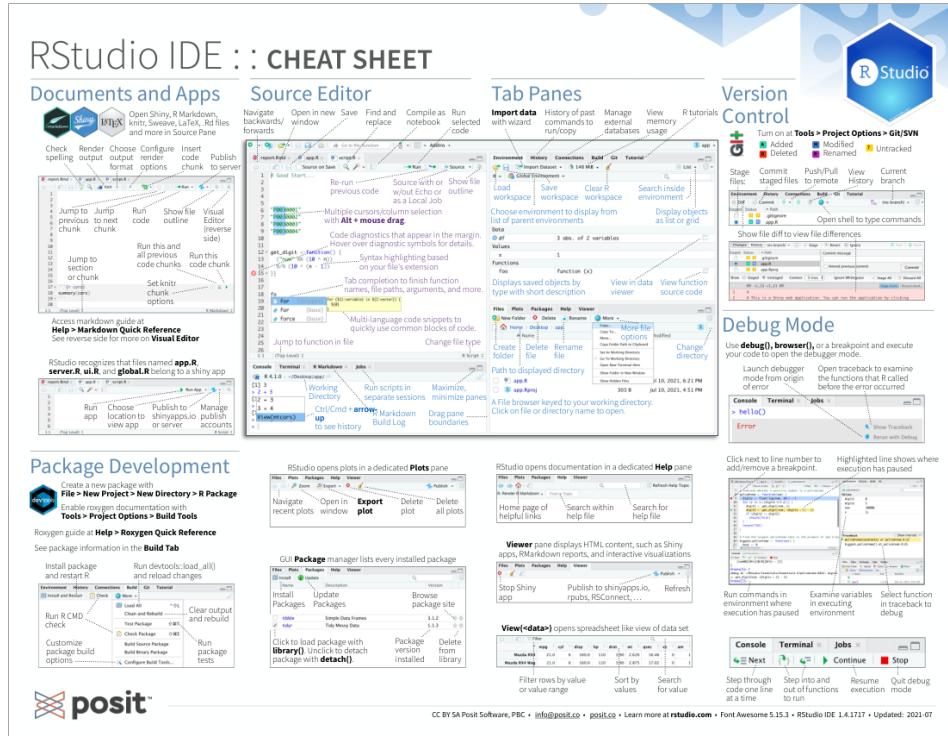




# RStudio cheatsheet

Use the cheatsheet to find your way in RStudio:

<https://github.com/rstudio/cheatsheets/blob/main/rstudio-ide.pdf>



## Working with R in RStudio

1. **Editor** (top left): Press the **Run** button or select code and press **CTRL + Enter** (**⌘ + Enter**).

- Analysis script
- Reproducibility

2. **Console** (bottom left): Simply press **Enter**.

- R as a calculator
- Trying out things before adding to the editor

The result of your command(s) will appear in the tab **Console** if the commands are intended to print something, and/or in the tab **Plots** if the commands generate a plot.

## R as a calculator

In the **Console**, R can simply be used as a calculator:

```
1 2 + 3
```

```
[1] 5
```

```
1 2 * 4
```

```
[1] 8
```

```
1 2^5
```

```
[1] 32
```

```
1 6 / 2 * (1 + 2)
```

```
[1] 9
```

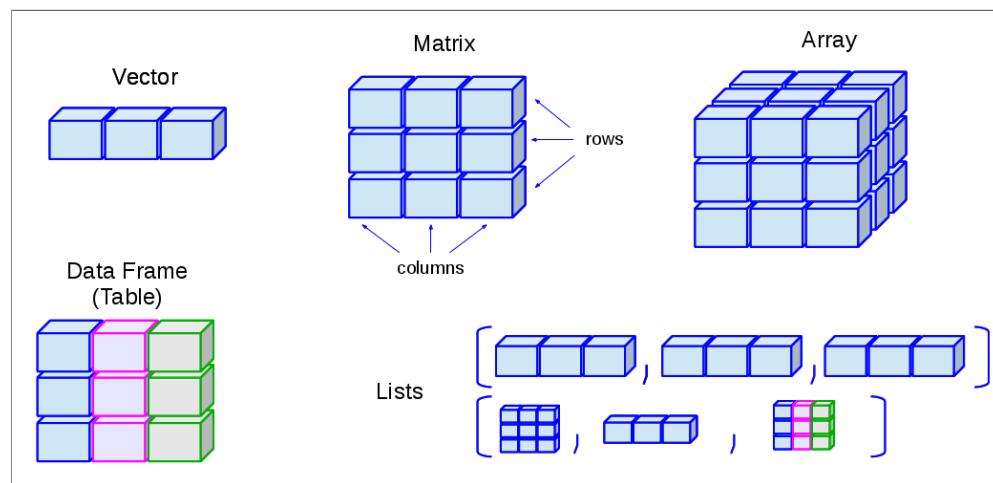
## Commenting in scripts

It is recommended to comment your scripts using #:

```
1 # This script illustrates R as a calculator
2
3 6 / 2 * (1 + 2) # Comments can also be placed to the right of code.
```

## Objects in R

Objects can be a single piece of data (e.g., `3.14` or `"Bern"`), or they can consist of structured data.



## Object classes

All the objects stored in R have a class which tells R how to handle the object. There are many possible classes, but common ones include:

Class	Description	Example(s)
numeric	Any real number	1, 3.14, 8.8e6
character	Individual characters or strings, quoted	"a", "Hello, World!"
factor	Categorical/qualitative variables	Ordered values of economic status
logical	Boolean variables	TRUE and FALSE
Date/POSIXct	Calendar dates and times	"2023-06-05"

Other object classes are `array`, `data.frame`, `list`, and `tibble` (similar to `data.frame`).

## Assigning values to objects

R uses `<-` (shown in text blocks here as `←`) to assign values to an object name (you might also see `=` used, but this is not best practice). Object names are case-sensitive, i.e., `X` and `x` are different.

```
1 x ← 2  
2 x
```

```
[1] 2
```

```
1 x * 4
```

```
[1] 8
```

```
1 x + 2
```

```
[1] 4
```

## Combine values

The function `c()` combines/concatenates single R objects into a vector (or list) of R objects:

```
1 x <- c(1, 4, 6, 8)
2 x
```

```
[1] 1 4 6 8
```

```
1 sum(x) # sum() is another function that returns the sum of vector elements
```

```
[1] 19
```

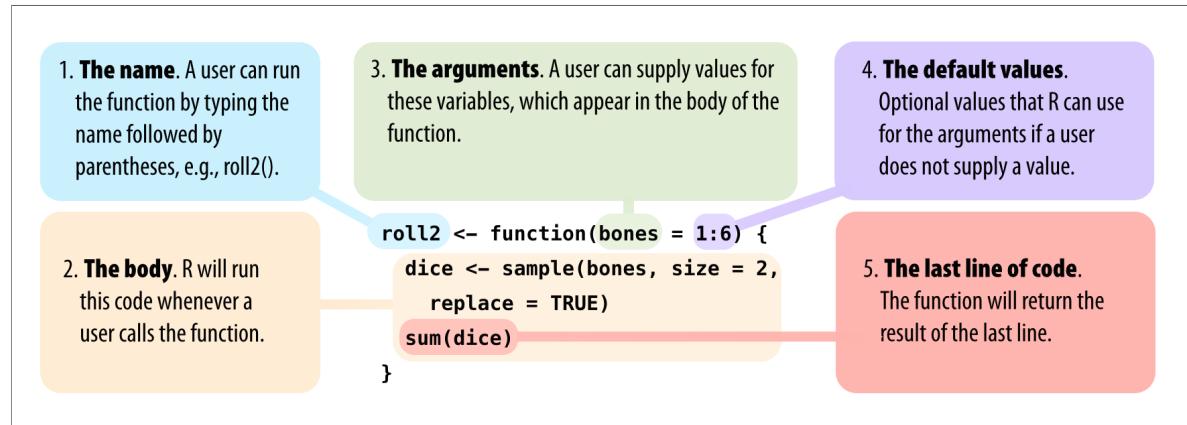
You can perform functions to entire vectors of numbers very easily.

```
1 x + 5
```

```
[1] 6 9 11 13
```

# Writing your own functions

Every function in R has three basic parts: a name, a body of code, and a set of arguments. To make your own function, you need to replicate these parts and store them in an R object, which you can do with the function `function()`.



## The R ecosystem

The build-in functionality of Base R can be expanded with **packages** that others have developed and published.

The Comprehensive R Archive Network (**CRAN**) has been the main source of R packages. Nowadays, GitHub also contains many packages and has arguably become the primary location for package development. Some packages, e.g., **tidyverse**, are so-called meta-packages - they load a collection of other packages.

Install new packages as follows:

```
1 install.packages("packagename")
```

Packages must be loaded each R session to give access to their functionality:

```
1 library(packagename)
```

## Getting help with R

The `help()` function and `?` help operator in R provide access to the documentation pages for R functions, data sets, and other objects, both for packages in the standard R distribution and for contributed packages.

Cheat sheets exist for many packages and topics:

<https://rstudio.github.io/cheatsheets>

Help for R is abundant. 99.9% of your questions will have been asked before, so Google is your friend, or ask on Twitter/Mastodon using the `#rstats` tag.

# Ask ChatGPT

CH

Write a function in R that calculates the geometric mean of a vector.



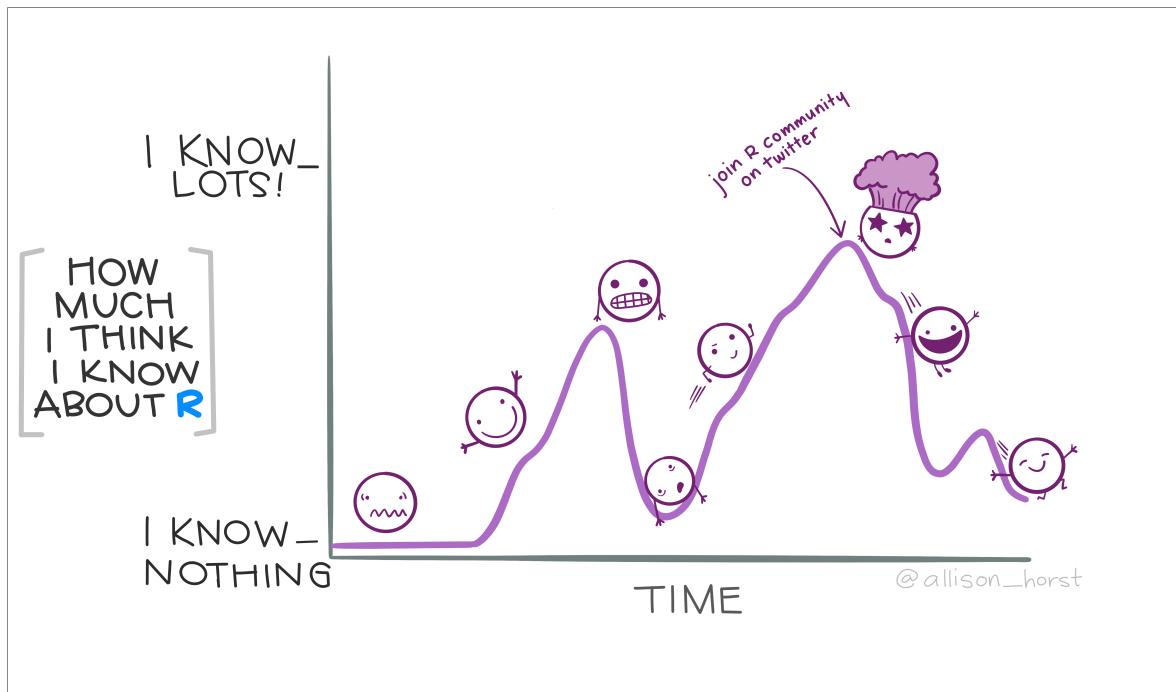
Certainly! Here's an example of a function in R that calculates the geometric mean of a vector:

R Copy code

```
geometric_mean <- function(vector) {  
  n <- length(vector)  
  if (n == 0) {  
    stop("Vector must not be empty.")  
  }  
  prod <- prod(vector)  
  gm <- prod^(1/n)  
  return(gm)  
}
```

This function takes a vector as input and calculates its geometric mean by raising the product of the vector's elements to the power of 1 divided by the number of elements in the vector.

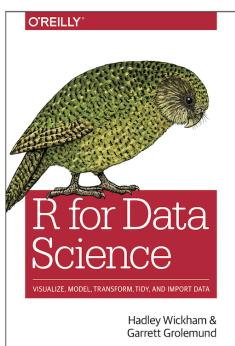
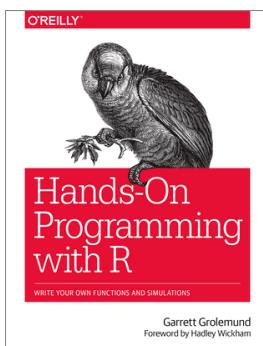
## The Rollercoaster



## Further resources

There are numerous online tutorials and books on R, RStudio with specific applications to epidemiology, public health, and data science:

- [The R Manuals](#)
- [Hands-On Programming with R](#)
- [The Epidemiologist R Handbook](#)
- [Introduction to R for Public Health Researchers](#)
- [R for Data Science](#)



[Anita.gerfridsson@chalmers.se](mailto:Anita.gerfridsson@chalmers.se) [The Epidemiologist R Handbook](#)

# R projects

## Why use projects in R(Studio)?

---

The use of projects (`.Rproj` files) is fundamental to organized coding and project management. There are four main reasons why you should use projects essentially 100% of the time while using RStudio:

- It can take less than 30 seconds to set up.
- It keeps all relevant files in the same place.
- It sets the working directory, so you can use relative paths.
- It allows for version control (see tomorrow).

## Folder structures

Using consistent folder structures across projects helps you work more efficiently. An example of the folder structure for a project looks like this:

- R
  - 00\_main.R
  - 01\_cleaning.R
  - 02\_analysis.R
  - 03\_plotting.R
- data
  - processed
    - cleaned\_data.csv
    - processed\_data.rds
  - raw
    - original\_data.csv
    - spreadsheet\_data.xlsx
- output
  - figures
    - 01\_figure.png
    - 02\_figure.pdf
  - tables
    - 01\_table.csv
    - 02\_table.rds
- products
  - manuscript
    - manuscript.docx
    - manuscript.html
    - manuscript.pdf
    - manuscript.qmd
  - report
    - report.html
    - report.qmd
  - slides

- slides.html
- slides.qmd
- .gitignore
- README.md
- project-template.Rproj

# **Naming files**

## **Bad examples**

- myabstract.docx
- Long file name using “spaces” & punctuation!.xlsx
- figures 1.png
- SDEFI7\_jknsfol.txt

## **Better**

- 2023-02-15\_abstract-conference-X.docx
- still-long-but-no-punctuation-or-spaces.xlsx
- fig01\_scatter-mpg-vs-vol.png
- more-meaningful-name.txt

# in that case...



## Naming R files

Analysis scripts can get loooooooong... Don't be afraid to break them up into smaller chunks.

Use sequential numbers and descriptive names, e.g.:

- `01_cleaning.R` cleans your data,
- `02_analysis.R` performs your analysis,
- `03_plotting.R` plots the results of your analysis.

Sequential numbers allow you to sort the files according to the sequence in which you run them.

Descriptive names inform you of what is actually in there.

You can use a main or master file (`00_main.R`) to run all other files and create a reproducible analysis.

Also see [\*\*How to name files\*\*](#) from Jennifer Bryan and [\*\*The tidyverse style guide\*\*](#).

## Add README files

Adding `README.md` (Markdown) or `README.txt` (plain text) files to your project folder and subfolders can be useful to describe your project and/or the content of folders, and provide instructions.

Tomorrow, you will learn more about Markdown (and Quarto).

## Exercise 1: Create a project

We have set up a template project for you, including a directory structure:

<https://github.com/ISPMBern/project-template>

1. Download the template (click Code and then Download ZIP).
2. Unzip the file to a suitable directory on your computer.
3. Rename the folder to something more suitable.
4. Rename the `.Rproj` file to something more meaningful to you (the same as the folder?).
5. Open the project in RStudio (double click the icon)

You will use this project for the rest of the course...

## Required packages for the next exercises

To work on the next exercises, you have to install the following packages:

- `usethis` - Workflow package
- `gitcreds` - Queries Git credentials from R
- `here` - Easy file referencing
- `tidyverse` - A set of packages
- `medicaldata` - Medical data sets
- `cowplot` - Features to create publication-quality figures

Simply type `install.packages("packagename")`, but RStudio will ask you about it as well if you want to load a package that you haven't installed yet.

Artwork by [@allison\\_horst](#)

# **Tidyverse and data wrangling**

## **Base R**

Collection of ca 25 packages that have been developed since R's conception (ca 25 years ago)

This age is often evident in the syntax - inconsistent option names and/or ordering of options, sometimes possible to tell which features were afterthoughts

Syntax varies widely across add-on packages

Enter the tidyverse...

## What is the tidyverse?



A group of R packages designed in a consistent manner along the principles of *tidy data*  
Primarily contains packages for data import (**readr**), manipulation (“wrangling”; **dplyr**, **forcats**, **stringr**) and visualization (**ggplot2**)

## What is the tidyverse?



A group of R packages designed in a consistent manner along the principles of *tidy data*

Primarily contains packages for data import (`readr`), manipulation (“wrangling”; `dplyr`, `forcats`, `stringr`) and visualization (`ggplot2`)  
Load the whole thing via `library(tidyverse)` or individual packages as usual (e.g. `library(ggplot2)`)

## Tidy data

Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

## Tidy data

Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

## Tidy data

Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

## Making small(!!) datasets by hand

Use case - look up tables.

In base R:    With the tidyverse:

```
1 data.frame(code = c(0, 1),
2             label = c("male", "female"))
```

```
code  label
1    0   male
2    1 female
```

Fine very small datasets, but unwieldy with many variables and/or many observations

```
1 tibble::tribble(
2   ~code, ~label,
3   0,      "male",
4   1,      "female"
5 )
```

```
# A tibble: 2 × 2
  code label
  <dbl> <chr>
1     0 male
2     1 female
```

Much nicer... immediately clear which label belongs to which code

No functional difference between a tibble and a dataframe, just a slightly different print method

## Getting data into R

R has a wide range of tools for importing data.

Base R

- `read.csv`
  - `read.csv2`
  - `read.delim`
- `tidyverse (readr)`
- `read_csv`
  - `read_csv2`
  - `read_delim`

## Getting data into R

R has a wide range of tools for importing data.

Base R

- `read.csv`
- `read.csv2`
- `read.delim`

tidyverse (`readr`)

- `read_csv`
- `read_csv2`
- `read_delim`

Others

- `readxl:: read_xlsx`
- `REDCapR:: redcap_read`
- `haven :: read_spss`

## Getting data into R

R has a wide range of tools for importing data.

Base R

- `read.csv`
- `read.csv2`
- `read.delim`

tidyverse (`readr`)

- `read_csv`
- `read_csv2`
- `read_delim`

Others

- `readxl:: read_xlsx`
- `REDCapR:: redcap_read`
- `haven :: read_spss`

And even more

- `secuTrialR:: read_secuTrial`
- `odbc :: dbConnect`
- `httr2`

## Data sometimes already exists in R...

Many published datasets already exist in R, either in the basic installation or via a package

```
1 data(mtcars)
2 data(iris) # very popular in examples
```

From packages, e.g. **medicaldata**<sup>1</sup>

```
1 install.packages("medicaldata")
2 library(medicaldata)
3 strep_tb
```

```
# A tibble: 107 × 13
  patient_id arm      dose_strep_g dose_PAS_g gender baseline_condition
  <chr>       <fct>    <dbl>     <dbl>   <fct>        <fct>
1 0001        Control     0          0 M      1_Good
2 0002        Control     0          0 F      1_Good
3 0003        Control     0          0 F      1_Good
4 0004        Control     0          0 M      1_Good
5 0005        Control     0          0 F      1_Good
6 0006        Control     0          0 M      1_Good
7 0007        Control     0          0 F      1_Good
8 0008        Control     0          0 M      1_Good
9 0009        Control     0          0 F      2_Fair
10 0010       Control     0          0 M      2_Fair
# i 97 more rows
# i 7 more variables: baseline_temp <fct>, baseline_esr <fct>,
#   baseline_cavitation <fct>, strep_resistance <fct>, radiologic_6m <fct>,
#   rad_num <dbl>, improved <lgl>
```

## Getting data into R in practice

We place the dataset in the appropriate folder (`01_original_data`) and read it in with the appropriate function (e.g. `read_csv`)

```
1 library(readr)
2 data <- read_csv(here("data", "raw", "MyData.csv"))
```

Depending on the file, you might need `read_csv2`, which is configured for e.g. German environments where CSVs are actually semi-colon (;) separated, because the comma is used in numbers . . .

In base R

```
1 data <- read.csv(here("data", "raw", "MyData.csv"))
```

Virtually identical... `readr` is slightly faster and automatically converts some variable types

# Getting data into R in practice

More common(?): excel files...

```
1 library(readxl) # informal tidyverse member
2 data <- read_xlsx(here("data", "raw", "MyData.xlsx"))
```

Once you've loaded a dataset, it's good practice to inspect the data to see that it's loaded correctly

```
1 str(strep_tb)
```

```
tibble [107 x 13] (S3:tbl_df/tbl/data.frame)
$ patient_id      : chr [1:107] "0001" "0002" "0003" "0004" ...
$ arm             : Factor w/ 2 levels "Streptomycin",..: 2 2 2 2 2 2 2 2 2 ...
$ dose_strep_g    : num [1:107] 0 0 0 0 0 0 0 0 0 ...
$ dose_PAS_g     : num [1:107] 0 0 0 0 0 0 0 0 0 ...
$ gender          : Factor w/ 2 levels "F","M": 2 1 1 2 1 2 1 2 1 ...
$ baseline_condition: Factor w/ 3 levels "1_Good","2_Fair",..: 1 1 1 1 1 1 1 1 2 ...
$ baseline_temp   : Factor w/ 4 levels "1_98-98.9F","2_99-99.9F",..: 1 3 1 1 2 3 2 2 4 ...
$ baseline_esr    : Factor w/ 4 levels "1_0-10","2_11-20",..: 2 2 3 3 3 3 3 3 3 ...
$ baseline_cavitation: Factor w/ 2 levels "no","yes": 2 1 1 1 1 1 2 2 2 ...
$ strep_resistance: Factor w/ 3 levels "1_sens_0-8","2_mod_8-99",..: 1 1 1 1 1 1 1 1 1 ...
$ radiologic_6m   : Factor w/ 6 levels "6_Considerable_improvement",..: 1 2 2 2 2 1 2 2 2 ...
$ rad_num         : num [1:107] 6 5 5 5 6 5 5 5 5 ...
$ improved        : logi [1:107] TRUE TRUE TRUE TRUE TRUE TRUE ...
```

## XLSX vs CSV vs database (e.g. REDCap)

### Watch Darren Dahly open an Excel file

Excel is easy to use, accessible, ...

...but it tries to be clever (e.g. formatting dates, gene names (30% of genetics papers contain mangled names in tables))

CSV can be created by Excel and other software (common export format from databases), has less issues with formatting

Both lack data validation protocols (more control over entered data)

Databases provide data validation, many are Human Research Act compliant (MS Access is not), many have ways to export data directly from the database to R, or simple ways to import exports

## What do you think this is?

```
[1] "\xfc" "\xe4" "\xe9" "\xe0"
```

# Dealing with special characters

```
[1] "\xfc" "\xe4" "\xe9" "\xe0"
```

```
[1] "ü" "ä" "é" "à"
```

Common issue when working in Switzerland - ä, ö, ü, é, è, à, etc

File encoding influences exactly how (special) characters are represented in a file

R needs that information to make sense of the data

```
1 # base
2 read.csv("path/to/file.csv", fileEncoding = "UTF-8")
3 # tidyverse (readr)
4 read_csv("path/to/file.csv", fileEncoding = "UTF-8",
5           locale = locale(encoding = "UTF-8"))
```

Free text, check the encoding!

Pro-tip: use Notepad++ to discover the encoding used, and possibly to convert to a different encoding (saving it to a different file...)

Use English as much as possible - saves time dealing with special characters AND no need to translate tables for publications

## Your turn!

Read file [insurance\\_with\\_date.csv](#) into R and explore it a little.

---

- How many observations and variables does it have?
- What types of variables does it include?
- What difference is there when importing via tidyverse ([readr\\$ read\\_csv](#)) vs base-R ([read.csv](#))?

You have 5 minutes... go!

# Solution

```
1 library(readr)
2 dat <- read_csv("data/raw/insurance_with_date.csv")
3 str(dat)
```

```
spc_tbl_ [1,338 x 9] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
$ X      : num [1:1338] 1 2 3 4 5 6 7 8 9 10 ...
$ age    : num [1:1338] 59 24 28 22 60 38 51 44 47 29 ...
$ sex    : chr [1:1338] "male" "female" "female" "male" ...
$ bmi    : num [1:1338] 31.8 22.6 25.9 25.2 36 ...
$ children: num [1:1338] 2 0 1 0 0 3 0 0 1 2 ...
$ smoker : chr [1:1338] "no" "no" "no" "no" ...
$ region : chr [1:1338] "southeast" "southwest" "northwest" "northwest" ...
$ charges : num [1:1338] 13086 2574 4411 2321 13435 ...
$ date   : Date[1:1338], format: "2001-01-15" "2001-01-17" ...
- attr(*, "spec")=
.. cols(
..   X = col_double(),
..   age = col_double(),
..   sex = col_character(),
..   bmi = col_double(),
..   children = col_double(),
..   smoker = col_character(),
..   region = col_character(),
..   charges = col_double(),
..   date = col_date(format = ""))
.. )
- attr(*, "problems")=<externalptr>
```

```
1 dat2 <- read.csv("data/raw/insurance_with_date.csv")
2 str(dat2)
```

```
'data.frame': 1338 obs. of  9 variables:
$ X      : int  1 2 3 4 5 6 7 8 9 10 ...
$ age    : int  59 24 28 22 60 38 51 44 47 29 ...
$ sex    : chr "male" "female" "female" "male" ...
$ bmi    : num  31.8 22.6 25.9 25.2 36 ...
$ children: int  2 0 1 0 0 3 0 0 1 2 ...
$ smoker : chr "no" "no" "no" "no" ...
$ region : chr "southeast" "southwest" "northwest" "northwest" ...
$ charges : num  13086 2574 4411 2321 13435 ...
$ date   : chr "2001-01-15" "2001-01-17" "2001-01-22" "2001-01-29" ...
```

## 1. currently contains 19 published medical datasets

# Data types

## Text/character/string - in R, **character**

```
1 sex <- c("male", "female", "female", "male")
```

cannot (always) be used models, generally needs converting to another format

## Encoded categorical information - factors

from a known list of options...

```
1 fct <- factor(sex, levels = c("male", "female", "non-binary"))
2 fct
```

```
[1] male   female female male
Levels: male female non-binary
```

from encoded data...

```
1 factor(c(0, 1), levels = 0:2, labels = c("male", "female", "non-binary"))
```

```
[1] male   female
Levels: male female non-binary
```

factors are more suitable for models than text

## Numbers - numeric

```
1 num ← 1:4  
2 num
```

```
[1] 1 2 3 4
```

```
1 c(1,1,2,3)
```

```
[1] 1 1 2 3
```

```
1 seq(1, 10, 2)
```

```
[1] 1 3 5 7 9
```

```
1 seq(1, 10, length.out = 4)
```

```
[1] 1 4 7 10
```

```
1 num * num
```

```
[1] 1 4 9 16
```

```
1 num * 2
```

```
[1] 2 4 6 8
```

```
1 num %*% num # matrix multiplication (scalar product)
```

```
[1,] [,1]  
[1,] 30
```

## Binary/Boolean/Yes/No variables - logical

```
1 TRUE
```

```
[1] TRUE
```

```
1 (lgl ← c(TRUE, FALSE, FALSE, TRUE))
```

```
[1] TRUE FALSE FALSE TRUE
```

```
1 num > 2
```

```
[1] FALSE FALSE TRUE TRUE
```

is a condition true or false, yes or no, e.g. death

## Keeping information together - **dataframes**

For when all things the same length...

```
1 df ← data.frame(sex = sex,
2                   fct = fct,
3                   num = num,
4                   lgl = lgl)
5 tibble::tibble(sex = sex,
6                 fct = fct,
7                 num = num,
8                 lgl = lgl)
```

```
# A tibble: 4 × 4
  sex     fct      num   lgl
  <chr>   <fct>   <int> <lgl>
1 male    male     1  TRUE
2 female  female   2 FALSE
3 female  female   3 FALSE
4 male    male     4  TRUE
```

# Keeping information together - **lists**

For when the objects have different lengths...

```
1 lst ← list(letter = "a",
2             numbers = rnorm(10),
3             data = df)
4 lst
```

```
$letter
[1] "a"

$numbers
[1] -1.125047263 -0.225204809  1.490237237  0.029200289 -0.064183197
[6] -1.788784925 -0.324374871 -1.022287549 -0.752372381 -0.005636146

$data
  sex   fct num lgl
1 male  male  1 TRUE
2 female female  2 FALSE
3 female female  3 FALSE
4 male  male  4 TRUE
```

Getting elements out again is the same for both

```
1 df$sex
```

```
[1] "male"  "female" "female" "male"
```

```
1 lst$letter
```

```
[1] "a"
```

## Piping: %>%, |>

When googling, you will encounter pipes. They enable chaining operations together.

Two main varieties:

%>% is from the `magrittr` package, introduced ca 2014

|> (shown in text blocks here as ▷) was added to base R in 2021 (v4.1.0)

```
1 data ▷  
2   mutate(new_var = rnorm(10)) ▷  
3   rename(random = new_var) ▷  
4   etc()
```

Especially useful in data wrangling...

## Without pipes

Essentially the same code as the last slide, just in base R  
Nesting calls...

```
1 etc(
2   rename(
3     mutate(data,
4       new_var = rnorm(10)),
5     random = new_var
6   )
7 )
```

...or saving intermediate objects...

```
1 tmp <- mutate(data, new_var = rnorm(10))
2 tmp <- rename(tmp, random = new_var)
3 etc(tmp)
```

## Data wrangling with `dplyr` and the tidyverse

Most of the tidyverse uses verbs as their function names...

- `readr :: read_csv` reads a Comma-Separated-Value file
- `dplyr :: filter` keep observations matching some criteria
- `dplyr :: mutate` modifies the data (change existing, add new variables)
- `dplyr :: rename` renames variables
- `stringr :: str_detect` detects whether the first `string` contains a particular piece of text (regular expression - regex)

```
1 library(dplyr)
```

## Selecting variables

The tidyverse offers a range of methods to select variables and these methods are used in many of the functions that we will discuss.

```
1 strep_tb %> select(patient_id, arm, gender)
2 strep_tb %> select(patient_id:gender, last_col())
3 strep_tb %> select(1:2, 13)
4 vars <- c("patient_id", "arm", "gender")
5 strep_tb %> select(all_of(vars))
```

### Base R examples

```
1 strep_tb[, c("patient_id", "arm", "gender")]
2 strep_tb[, c(1, 2, 13)]
3 vars <- c("patient_id", "arm", "gender")
4 strep_tb[, vars]
```

## Selecting variables

Also by variable class or aspects of the variable name

```
1 strep_tb %> select(where(is.factor))
2 strep_tb %> select(contains("i")) # contains a literal string
3 strep_tb %> select(matches("ll")) # matches a regex
4 strep_tb %> select(starts_with("b"))
```

More tricky with base R

```
1 strep_tb[, sapply(strep_tb, is.factor)]
2 strep_tb[, grepl("i", names(strep_tb))]
3 strep_tb[, grepl("^b", names(strep_tb))]
```

## Selecting observations

filtering

```
1 strep_tb %> filter(arm = "Control")
2 strep_tb %> filter(dose_strep_g > 0)
3
4 strep_tb %> filter(improved)
```

With base R

```
1 strep_tb[strep_tb$arm == "Control", ]
2 strep_tb[strep_tb$dose_strep_g > 0, ]
3 strep_tb[strep_tb$improved, ]
4 subset(strep_tb, improved)
```

# Selecting observations

## slicing

```
1 # specific rows
2 strep_tb %> slice(1, 2, 3, 5, 8, 13)
3
4 # first 10
5 strep_tb %> slice_head(n = 10)
6 strep_tb %> slice_head(n = -10) # exclude first 10 rows
7
8 # first 10% of observations
9 strep_tb %> slice_head(prop = 0.1)
10
11 # last 10
12 strep_tb %> slice_tail(n = 10)
```

Not so different with base R...

```
1 strep_tb[c(1, 2, 3, 5, 8, 13), ] # specific rows
2 strep_tb[1:10, ] # first 10
3 head(strep_tb, n = 10) # or strep_tb %> head(n = 10)
4 strep_tb[(nrow(strep_tb)-10):nrow(strep_tb), ] # last 10
5 tail(strep_tb, n = 10) # or strep_tb %> tail(n = 10)
```

Pay attention to ordering in the dataframe

# Modifying data

`mutate` is your friend

```
1 strep_tb >
2   mutate(dose_strep_g = dose_strep_g + 2)
```

Very simple in base too...

```
1 strep_tb$dose_strep_g <- strep_tb$dose_strep_g + 2
```

Not restricted to a single change

```
1 strep_tb >
2   mutate(dose_strep_g = dose_strep_g + 2,
3         control = arm == "Control")
```

In base R, something similar can be done with the rarely used `within` function

```
1 within(strep_tb, {
2   dose_strep_g <- dose_strep_g + 2
3   control <- arm == "Control"
4 })
```

```
# A tibble: 107 × 14
  patient_id arm      dose_strep_g dose_PAS_g gender baseline_condition
  <chr>        <fct>          <dbl>       <dbl> <fct>    <fct>
1 0001        Control         2           0 M      1_Good
2 0002        Control         2           0 F      1_Good
3 0003        Control         2           0 F      1_Good
4 0004        Control         2           0 M      1_Good
5 0005        Control         2           0 F      1_Good
6 0006        Control         2           0 M      1_Good
7 0007        Control         2           0 F      1_Good
8 0008        Control         2           0 M      1_Good
9 0009        Control         2           0 F      2_Fair
10 0010       Control         2           0 M      2_Fair
# i 97 more rows
# i 8 more variables: baseline_temp <fct>, baseline_esr <fct>,
#   baseline_cavitation <fct>, strep_resistance <fct>, radiologic_6m <fct>,
#   rad_num <dbl>, improved <lgl>, control <lgl>
```

## Same change to many variables

add two to all `dose_*` variables

```
1 strep_tb >
2   mutate(across(starts_with("dose"), ~ .x + 2))
```

creating new variables

```
1 strep_tb >
2   mutate(across(starts_with("dose"), ~ .x + 2, .names = "{.col}_plus2"))
```

more generic, using variable class

```
1 strep_tb >
2   mutate(across(where(is.numeric), ~ .x + 2, .names = "{.col}_plus2"))
```

Not useful here, but handy for e.g. factors (examples later)

One possible base method

```
1 for(var in c("dose_strep_g", "dose_PAS_g")){
2   strep_tb[, paste0(var, "_plus2")] ← strep_tb[, var] + 2
3 }
```

# Conditional modifications

Sometimes you need to do something under one circumstance, something else under another

none   if else   case when   Base R

all individuals have short measurements

```
1 strep_tb >
2   mutate(dose_strep_g_corr = dose_strep_g + 2)
```

create text for male/female

```
1 strep_tb >
2   mutate(txt = if_else(gender == "M",
3                         # when TRUE
4                         "Male",
5                         # when FALSE
6                         "Female"))
```

We want some specific text in some cases

```
1 strep_tb >
2   mutate(dose_strep_g_corr = case_when(
3     # Male, Streptomycin
4     gender == "M" & arm == "Streptomycin" ~ "M Strepto",
5     # Female, Streptomycin
6     gender == "F" & arm == "Streptomycin" ~ "F Strepto",
7     # all others are OK
8     TRUE ~ "Control")
9   )

1 # initialize the variable
2 strep_tb$txt <- "Control"
3
4 # replace the values for males in the streptomycin group
5 strep_tb$txt[
6   with(strep_tb, gender == "M" & arm == "Streptomycin") # which cases
7 ] <- "M Strepto"
8
9
10 # replace the values for females in the streptomycin group, using a temporary variable
11 to_change <- with(strep_tb, gender == "F" & arm == "Streptomycin")
12
13 strep_tb$txt[to_change] <- "F Strepto"
14
15 # strep_tb$x[to_change] <- strep_tb$x[to_change] * 2
16
17 rm(to_change) # clean up
```

hmmm... tidyverse syntax is much nicer!?

## Working with strings: `stringr`

The `stringr` package contains functions specifically for working with strings.

Most functions start with `str_`.

```
1 library(stringr)
2 txt <- "A silly exAmple "
```

### Changing case

```
1 str_to_lower(txt)
```

```
[1] "a silly example "
```

```
1 str_to_sentence(txt)
```

```
[1] "A silly example "
```

```
1 str_to_title(txt)
```

```
[1] "A Silly Example "
```

```
1 str_to_upper(txt)
```

```
[1] "A SILLY EXAMPLE "
```

### Lengths

```
1 str_length(txt)
```

```
[1] 16
```

```
1 str_count(txt, "\w+")
```

```
[1] 3
```

# Working with strings: `stringr`

## Remove white space

```
1 str_squish(txt)
```

```
[1] "A silly exAmple"
```

## Substrings

```
1 str_sub(txt, 3)
```

```
[1] "silly exAmple "
```

```
1 str_sub(txt, 3, -4)
```

```
[1] "silly exAmp"
```

```
1 word(txt, 2)
```

```
[1] "silly"
```

## Replacements

```
1 str_replace(txt, "e", "XX")
```

```
[1] "A silly XXxAmple "
```

```
1 str_replace_all(txt, "e", "XX")
```

```
[1] "A silly XXxAmp!XX "
```

# Working with strings: `stringr`

## Detect a substring

```
1 str_detect(txt, "x")
```

```
[1] TRUE
```

```
1 str_detect(txt, "z")
```

```
[1] FALSE
```

## Splitting

```
1 str_split(txt, " ")
```

```
[[1]]  
[1] "A"      "siLLy"    "exAmple"  ""
```

## Using `stringr` within `mutate`

```
1 strep_tb >  
2   mutate(txt = as.character(baseline_condition),  
3         upper = str_to_upper(txt),  
4         good = str_detect(txt, "Good"),  
5         no_number = str_replace(txt, "^[[:digit:]]_ ", "")) >  
6   select(txt:no_number) > unique()
```

```
# A tibble: 3 × 4  
  txt    upper  good  no_number  
  <chr>  <chr>  <lgl> <chr>  
1 1_Good 1_GOOD TRUE  Good  
2 2_Fair 2_FAIR FALSE Fair  
3 3_Poor 3_POOR FALSE Poor
```

## Working with factors: `forcats`

The `forcats` package contains functions specifically for working with factors.

Functions (almost) all begin with `fct_`.

```
1 fac ← strep_tb$baseline_condition[c(1, 15, 29)]  
2 fac
```

```
[1] 1_Good 2_Fair 3_Poor  
Levels: 1_Good 2_Fair 3_Poor
```

Reverse the levels (particularly useful when making plots)

```
1 library(forcats)  
2 fct_rev(fac)
```

```
[1] 1_Good 2_Fair 3_Poor  
Levels: 3_Poor 2_Fair 1_Good
```

```
1 # factor(fac,  
2 #           levels = rev(levels(fac)))
```

Changes the order levels are shown in e.g. tables

# Working with factors: `forcats`

## Change level names

```
1 fct_recode(fac,
2           # new = "old"
3           Good = "1_Good",
4           Fair = "2_Fair",
5           Poor = "3_Poor")
```

```
[1] Good Fair Poor
Levels: Good Fair Poor
```

```
1 # factor(as.character(fac),
2 #         levels = levels(fac),
3 #         labels = c("Good", "Fair", "Poor"))
```

## This is also possible by regular expression (regex)

```
1 fct_relabel(fac,
2             str_replace, # function to use
3             # additional arguments to the function
4             pattern = "[123]_", # regex for "starts with 1, 2 or 3 and is followed by _"
5             replacement = "" # replace with nothing
6             )
```

```
[1] Good Fair Poor
Levels: Good Fair Poor
```

```
1 # factor(as.character(fac),
2 #         levels = levels(fac),
3 #         labels = gsub("[123]_", "", levels(fac)))
```

# Working with factors: `forcats`

## Within `mutate`

```
1 strep_tb >
2   mutate(baseline_condition_new = fct_relabel(baseline_condition,
3                                             str_replace,
4                                             pattern = "^[[:digit:]]_",
5                                             replacement = ""))
6   select(baseline_condition, baseline_condition_new) >
7   str()
```

```
tibble [107 x 2] (S3: tbl_df/tbl/data.frame)
$ baseline_condition    : Factor w/ 3 levels "1_Good","2_Fair",..: 1 1 1 1 1 1 1 1 2 2 ...
$ baseline_condition_new: Factor w/ 3 levels "Good","Fair",..: 1 1 1 1 1 1 1 2 2 ...
```

## Do it to all factors

```
1 strep_tb >
2   mutate(across(where(is.factor),
3                 ~ fct_relabel(.x,
4                               str_replace,
5                               pattern = "^[[:digit:]]_",
6                               replacement = ""))) >
7   select(where(is.factor)) >
8   str()
```

```
tibble [107 x 8] (S3: tbl_df/tbl/data.frame)
$ arm          : Factor w/ 2 levels "Streptomycin",..: 2 2 2 2 2 2 2 2 2 2 ...
$ gender        : Factor w/ 2 levels "F","M": 2 1 1 2 1 2 1 2 1 2 ...
$ baseline_condition : Factor w/ 3 levels "Good","Fair",..: 1 1 1 1 1 1 1 2 2 ...
$ baseline_temp      : Factor w/ 4 levels "98-98.9F","99-99.9F",..: 1 3 1 1 2 3 2 2 2 4 ...
$ baseline_esr       : Factor w/ 4 levels "0-10","11-20",..: 2 2 3 3 3 3 3 3 3 3 ...
$ baseline_cavitation: Factor w/ 2 levels "no","yes": 2 1 1 1 1 1 2 2 2 2 ...
$ strep_resistance   : Factor w/ 3 levels "sens_0-8","mod_8-99",..: 1 1 1 1 1 1 1 1 ...
$ radiologic_6m     : Factor w/ 6 levels "Considerable_improvement",..: 1 2 2 2 2 1 2 2 2 2 ...
```

## Working with dates: `lubridate`

`lubridate` provides a comprehensive set of functions for working with dates and date-times

Dates come in many formats, `lubridate` handles them easily

```
1 library(lubridate)
2 ymd("2023-01-20")
```

```
[1] "2023-01-20"
```

```
1 ymd_hm("2023-01-20 10:15")
```

```
[1] "2023-01-20 10:15:00 UTC"
```

```
1 dmy("20 January 2023")
```

```
[1] "2023-01-20"
```

```
1 mdy("January 20 2023")
```

```
[1] "2023-01-20"
```

```
1 mdy("Januar 20 2023")
```

```
[1] NA
```

```
1 mdy("1 20 2023")
```

```
[1] "2023-01-20"
```

```
1 mdy("1 20 2023")
```

```
[1] "2023-01-20"
```

```
1 dmy("the 1st of May 2023 was a Monday")
```

```
[1] "2023-05-01"
```

## Working with dates: base

In base R, it's not so easy... (see `?strptime` for details)

```
1 as.Date("2023-01-20")
```

```
[1] "2023-01-20"
```

```
1 as.Date("20 Jan 2023", format = "%d %B %Y")
```

```
[1] "2023-01-20"
```

```
1 as.Date("01 20 2023", format = "%m %d %Y")
```

```
[1] "2023-01-20"
```

```
1 as.POSIXct("2023-01-20 10:15")
```

```
[1] "2023-01-20 10:15:00 CET"
```

Very specific to system settings (language)...

Unless you have e.g. “2023-01-20 10:15”, stick with `lubridate`...

## Working with dates: `lubridate`

We can do maths with date(-time)s...

```
1 date1 <- ymd("2023-01-20")
2 date2 <- ymd("2023-02-20")
3 diff <- date2 - date1
4 diff
```

Time difference of 31 days

```
1 str(diff)
```

```
'difftime' num 31
- attr(*, "units")= chr "days"
```

It's normally worth converting it to a number...

```
1 diff > as.numeric()
```

[1] 31

Add a certain number of months

```
1 date1 + months(2)
```

[1] "2023-03-20"

## Working with dates: `lubridate`

Extracting components of the date(-time)

```
1 datetime ← ymd_hm("2023-01-23 15:30")
2 year(datetime)
```

```
[1] 2023
```

```
1 month(datetime)
```

```
[1] 1
```

```
1 hour(datetime)
```

```
[1] 15
```

With base R:

```
1 format(datetime, "%Y") # year
2 format(datetime, "%m") # month
3 format(datetime, "%H") # hour
4 format(datetime, "%M") # minute
5 format(datetime, "%x") # date, d.m.y format
```

Check the cheat sheet for lots more of `lubridate` capabilities

## Working with dates: internals

```
1 ymd("2023-01-20") %> str()
```

```
Date[1:1], format: "2023-01-20"
```

```
1 ymd_hm("2023-01-20 10:15") %> str()
```

```
POSIXct[1:1], format: "2023-01-20 10:15:00"
```

Stored internally as numbers! This allows the maths operations to work

```
1 ymd("2023-01-20") %> as.numeric()
```

```
[1] 19377
```

```
1 ymd_hm("2023-01-20 10:15") %> as.numeric()
```

```
[1] 1674209700
```

What do those numbers mean?

They're days and milliseconds since an origin...

## Working with dates: the origin

When is that origin (timepoint 0)?

```
1 ymd("2023-01-20") - as.numeric(ymd("2023-01-20"))
```

```
[1] "1970-01-01"
```

```
1 ymd_hm("2023-01-20 10:15") - as.numeric(ymd_hm("2023-01-20 10:15"))
```

```
[1] "1970-01-01 UTC"
```

Days since 1st January 1970

Milliseconds since 1st January 1970

## Your turn!

Using the insurance data you loaded earlier...

- make factors out of the `sex`, and `region`
- make a logical indicator for “has more than 2 children” and “smokes”
- add 6 months to the `date` variable

## Solution

```
1 reformatted ← dat ▷  
2   mutate(  
3     across(c(sex, region), factor),  
4     # sex = factor(sex),  
5     # region = factor(region),  
6     gt2_children = children > 2,  
7     smokes = smoker == "yes",  
8     date_6m = date + months(6)  
9     # date_6m = date + 30.4 * 6  
10    )
```

```
tibble [1,338 × 12] (S3: tbl_df/tbl/data.frame)  
$ X           : num [1:1338] 1 2 3 4 5 6 7 8 9 10 ...  
$ age         : num [1:1338] 59 24 28 22 60 38 51 44 47 29 ...  
$ sex         : Factor w/ 2 levels "female","male": 2 1 1 2 1 1 1 1 2 2  
...  
$ bmi         : num [1:1338] 31.8 22.6 25.9 25.2 36 ...  
$ children    : num [1:1338] 2 0 1 0 0 3 0 0 1 2 ...  
$ smoker      : chr [1:1338] "no" "no" "no" "no" ...  
$ region      : Factor w/ 4 levels "northeast","northwest",..: 3 4 2 2 1  
4 4 2 4 2 ...  
$ charges     : num [1:1338] 13086 2574 4411 2321 13435 ...  
$ date        : Date[1:1338], format: "2001-01-15" "2001-01-17" ...  
$ gt2_children: logi [1:1338] FALSE FALSE FALSE FALSE TRUE ...  
$ smokes      : logi [1:1338] FALSE FALSE FALSE FALSE FALSE FALSE ...  
$ date_6m     : Date[1:1338], format: "2001-07-15" "2001-07-17" ...
```

## Pivoting datasets

Sometimes it's necessary to pivot data. E.g. All observations from an individual are on a single row. For our analysis we need them to be in a single variable. `pivot_longer` is the tool for the task.

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

```
1 pivot_longer(data, cols,
2                   names_to = "year",
3                   values_to = "cases")
```

## Pivoting datasets

The opposite, observations on rows to observations in columns, is **pivot\_wider**

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T



country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

```
1 pivot_wider(data,
2                   names_from = "type",
3                   values_from = "count")
```

In base R, both these scenarios are handled by **reshape**, the syntax and documentation thereof is confusing... which is why it's two separate functions in **dplyr**

# Merging datasets (joins)

```
1 (a <- data.frame(id = 1:3,  
2                     v1 = letters[1:3]))
```

```
  id v1  
1  1  a  
2  2  b  
3  3  c
```

```
1 (b <- data.frame(id = 2:4,  
2                     v2 = LETTERS[1:3]))
```

```
  id v2  
1  2  A  
2  3  B  
3  4  C
```

Left join (observations  
in the first dataframe)

```
1 a %>% left_join(b)
```

```
  id v1   v2  
1  1  a <NA>  
2  2  b    A  
3  3  c    B
```

Right join (observations  
in the second  
dataframe)

```
1 a %>% right_join(b)
```

```
  id   v1 v2  
1  2     b  A  
2  3     c  B  
3  4 <NA>  C
```

Inner join (observations  
in both)

```
1 a %>% inner_join(b)
```

```
  id v1 v2  
1  2  b  A  
2  3  c  B
```

# Merging datasets (joins)

Full join (all observations)

```
1 a %>% full_join(b)
```

	id	v1	v2
1	1	a	<NA>
2	2	b	A
3	3	c	B
4	4	<NA>	C

Full join with differing variable names to join on

```
1 a %>% full_join(b,  
2   by = join_by(id_a == id_b) # new syntax!  
3   # by = c("id_a" = "id_b") # older syntax  
4 )
```

In base R, use **merge** and combinations of **all.x** and **all.y** to specify the different join types, and **by.x** and **by.y** to specify the variables to join on

```
1 merge(a, b, all.x = TRUE, all.y = FALSE, by.x, by.y) # left  
2 merge(a, b, all.x = FALSE, all.y = TRUE, by.x, by.y) # right  
3 merge(a, b, all.x = FALSE, all.y = FALSE, by.x, by.y) # inner  
4 merge(a, b, all.x = TRUE, all.y = TRUE, by.x, by.y) # full
```

## Summarizing data

At some point, you will have to create summary data. `dplyr` can help with that too. `summarize` is the appropriate function.

```
1 strep_tb %>
2   summarize(n = n(),
3             min = min(rad_num),
4             median = median(rad_num),
5             mean = mean(rad_num),
6             max = max(rad_num),
7             )
```

```
# A tibble: 1 × 5
#>   n     min   median   mean   max
#>   <int> <dbl>    <dbl> <dbl>    <dbl>
1  107      1       5   3.93      1
```

Remember `across`? It comes in useful here too

```
1 strep_tb %>
2   summarize(n = n(),
3             across(c(rad_num, dose_strep_g),
4                      list(min = ~ min(.x, na.rm = TRUE),
5                           mean = mean,
6                           median = median,
7                           max = max),
8                      .names = "{.col}_{.fn}"),
9             )
```

```
# A tibble: 1 × 9
#>   n rad_num_min rad_num_mean rad_num_median rad_num_max dose_strep_g_min
#>   <int>        <dbl>       <dbl>        <dbl>       <dbl>        <dbl>
1  107          1         3.93        5          6            0
#> # i 3 more variables: dose_strep_g_mean <dbl>, dose_strep_g_median <dbl>,
#> #   dose_strep_g_max <dbl>
```

## Summarizing data

What about grouped summaries? Two options... `group_by()`...

```
1 strep_tb >
2   group_by(arm) >
3     summarize(n = n(),
4                 min = min(rad_num),
5                 median = median(rad_num),
6                 mean = mean(rad_num),
7                 max = max(rad_num),
8               )
```

... or `.by` (new syntax)

```
1 strep_tb >
2   summarize(n = n(),
3             min = min(rad_num),
4             median = median(rad_num),
5             mean = mean(rad_num),
6             max = max(rad_num),
7             .by = arm
8           )
```

```
# A tibble: 2 × 6
  arm          n  min median  mean  max
  <fct>    <int> <dbl> <dbl> <dbl> <dbl>
1 Control      52     1     3  3.13     1
2 Streptomycin 55     1     6  4.67     1
```

## Labelling variables

Variable names tend to be short - less typing, less chance of making a typo.  
But they're not very useful for tables... We can add labels to variables,  
which various packages know how to use

```
1 library(labelled)
2 strep_tb_lab <- strep_tb ▷
3   set_variable_labels(arm = "Treatment",
4                       dose_strep_g = "Dose of Streptomycin",
5                       rad_num = "Radiologic response",
6                       baseline_temp = "Temp. at baseline",
7                       improved = "Improvement in radiologic response")
```

Retrieve the labels again

```
1 var_label(strep_tb_lab$rad_num)
```

```
[1] "Radiologic response"
```

## **Publication type tables**

Options:

- create something yourself, based on the summaries above (e.g. reshape, format, etc)
- use a package to do it for you...

## Publication type tables

gtsummary is my package of choice

```
1 library(gtsummary)
2 strep_tb_lab >
3   select(arm, dose_strep_g, baseline_temp,
4         rad_num, improved) >
5  tbl_summary(by = arm,
6             type = c(
7               rad_num ~ "continuous"
8             )) >
9   add_overall()
```

Results are very customisable (see the help files). The package also provides support for model output.

Characteristic	Overall, N = 107 <sup>1</sup>	Streptomycin, N = 55 <sup>1</sup>	Control, N = 52 <sup>1</sup>
Dose of Streptomycin			
0	52 (49%)	0 (0%)	52 (100%)
2	55 (51%)	55 (100%)	0 (0%)
Temp. at baseline			
1_98-98.9F	7 (6.5%)	3 (5.5%)	4 (7.7%)
2_99-99.9F	25 (23%)	13 (24%)	12 (23%)
3_100-100.9F	32 (30%)	15 (27%)	17 (33%)
4_100F+	43 (40%)	24 (44%)	19 (37%)
Radiologic response	5.00 (2.00, 6.00)	6.00 (3.00, 6.00)	3.00 (1.00, 5.00)
Improvement in radiologic response	55 (51%)	38 (69%)	17 (33%)

<sup>1</sup> n (%); Median (IQR)