A Short Introduction to Working With Data in R

Jonathan Whiteley

2023-09-17

Prerequisites

- Access to a copy of the R software
  - Get it from *www.r-project.org*, or ask your system administrator.
- Tidyverse packages installed on the same system as R
  - Please run this command in R *before* the workshop:

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

- Download the workshop files, including these slides, data, and scripts.
  - The workshop assumes the same file structure as in the link above.
- Knowledge of common mathematical operations: arithmetic, logarithms, etc.
- Knowledge of basic R concepts, such as *variables*, *objects*, *operators*, *functions*, *packages*, etc.
  - This is covered in the first workshop: "A Gentle Introduction to R"

Section 1

Welcome

Pop Quiz

We will review these *at the end*, so you can see how much you have learned.

- If multiple packages have functions with the same name, how can you specify which one to use?
- Does `R` store data in memory or temporary files?
- What is the limit to the size of objects and datasets that can be loaded into `R`?
- TRUE or FALSE: `R` has rules and conventions for naming functions
- TRUE or FALSE: if you use one package from the `tidyverse`, you have to use all of them.

Answer in the chat:

What is your favourite emoji? Why do you like to use it so much?

Introductions

- Name
- Job title, role

If you are comfortable sharing:

- Pronouns
- A hobby or activity you enjoy

- What are you hoping to learn most in today's workshop?

Learning Objectives

- Load tabular data into R
- Explore data to check that it was loaded correctly
- Export data from R to external files
- Data frames
- Clean data
  - Re-arrange & modify rows
  - Add & change columns
  - Edit values systematically
  - Change data types
- Tidy data
  - Change the *shape* of a data frame
- Re-use code, reproducible results, automated reports
  - Scripts
  - R Markdown, R Notebooks

Disclaimer

- There is often more than one way to achieve a desired result in `R`
- Some are faster in certain situations
- Some require less code, or are easier to write as code
- Some are more portable (work on multiple systems)
- But there is rarely a single 'best way'.

This workshop focuses on a coherent approach, that can be learned more easily and extended as needed to tackle bigger problems.

Feel free to take what you learn here and experiment, or explore alternatives. Find what works for *you*.

Section 2

File Paths and The Working Directory

The Working Directory

- When working with external files, it helps to know the current *working directory*
  - Any paths supplied to R functions will be relative to this path.

```
getwd()
```

- You can change the working directory with this command:

```
setwd('path/to/a/directory')
```

File paths

- A *file path* is a *character string* that represents the location of a file in your system (computer and OS)
- The format of paths can depend on the operating system (OS)
  - Some use `"/"` to separate directories
    e.g., “`/dir/subdir`”
  - Windows uses `"\"`
    e.g., `"C:\\dir\subdir"`
    `R` uses this as an escape character in strings, and must be escaped itself in paths (`"\\"`)
    e.g., `"C:\\\\dir\\subdir"`

Paths in R

- R generally uses and understands "/" in paths, even on Windows.[1]
  - e.g., "C:/dir/subdir"
  - on Windows, it also understands Windows-style paths: e.g, "C:\\\\dir\\\\subdir"
- R also has platform-independent functions for manipulating paths, such as `file.path()`, which I will use in examples to make them as reproducible as possible.

[1] For the gory details, see section 14.2 "Filepaths" in "An Introduction to R" (`help.start()`), `?file.path`, and documentation for related functions.

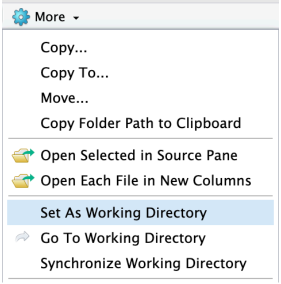
My paths are not like yours

- Directory (folder) names can also vary from one computer to another — it's difficult to show a path in this document that will also work on your computer!
- Once you set a working directory *on your computer* based on the structure of the files in this project, we can use *relative paths* that should also work on your computer (assuming your downloaded the workshop files in the same structure as provided).

Set the working directory

- For this workshop, set the working directory to location where you downloaded this presentation and accompanying files.
  - the directory that *contains* the folder named '`data`' that you downloaded along with the files for this workshop.
- Base `R` on Mac / Linux:
  - Menu item: "Misc > Change Working Directory..."
  - `CMD+D` on Mac; `CTL+D` on Linux (or Windows)
- In RStudio, you can use the *Files* pane (default bottom-right) to navigate to a directory in your system, and click on "*More > Set As Working Directory*"
  - or "*Session > Set Working Directory > To Files Pane Location*" in the RStudio menu.
- Base `R` on Windows:

```
setwd( choose.dir() )
```



Check your working directory

- Check to see that the working directory is in the right place, by checking to see if a known file exists (from R's perspective):

```
DF_path <- file.path("data", "data_example.csv")
file.exists(DF_path)
```

```
# [1] TRUE
```

- If the result of the statement above is not "TRUE" in your session, try one of the other approaches to change your working directory, and try again.

Section 3

Loading Data into R

csv files

- 'csv' = **C**omma **S**eparated **V**alues
  - files in this format have a '`.csv`' file extension.
- They are:
  - plain text files
  - used to represent tabular data, with each *row* on a line, and values in each *column* separated by commas (`,`)
  - readable by a wide variety of analysis software (highly portable)
  - simple—no embedded metadata
- We'll try to load this file into `R`:

  `example_data.csv`
  - *optional: you can try opening it in a text editor, or spreadsheet software, to see what's in the file.*

Load a csv file into R (basic)

```
?read.csv
read.csv(DF_path)
```

```
# Error in read.table(file = file, header = header, sep = sep, quote
#   more columns than column names
```

Load a csv file into R (basic)

```r
?read.csv
read.csv(DF_path)
```

```
# Error in read.table(file = file, header = header, sep = sep, quote
#   more columns than column names
```

- Uh oh! Something's not right.

Check the file contents

- Let's take a peek at the first few lines and see if we can identify the problem:

```r
readLines(DF_path, n = 4)
```

```
# [1] "Data from an experiment on the cold tolerance of the grass sp
# [2] "Modified from `data(CO2)`.  See `?CO2`."
# [3] "Type,Treatment,PlantNum,95,175,250,350,500,675,1000"
# [4] "Quebec,nonchilled,1,16,30.4,34.8,37.2,35.3,39.2,39.7"
```

Check the file contents

- Let's take a peek at the first few lines and see if we can identify the problem:

```r
readLines(DF_path, n = 4)
```

```
# [1] "Data from an experiment on the cold tolerance of the grass sp
# [2] "Modified from `data(CO2)`.  See `?CO2`."
# [3] "Type,Treatment,PlantNum,95,175,250,350,500,675,1000"
# [4] "Quebec,nonchilled,1,16,30.4,34.8,37.2,35.3,39.2,39.7"
```

- The first **2** lines don't look like comma-separated values!
- They look like extra information that is not part of the data table *structure*.

Load a csv file into R

- We can tell R to skip the lines with no data:
  - and we'll *assign* the result to a variable so we can work on it

```
DF <- read.csv(DF_path, skip = 2)
```

- Just because there were no Errors from R, doesn't mean there's nothing wrong with the data!

Section 4

Exploring Your Data

Object class: data frame

Before we explore our new data set, let's quickly review the kind of *object* we're dealing with:

```r
class(DF)
```

```
# [1] "data.frame"
```

```r
typeof(DF)
```

```
# [1] "list"
```

Data frames

head(): peek at the first few rows

```r
head(DF)
```

```
#     Type  Treatment PlantNum  X95 X175 X250 X350
# 1 Quebec nonchilled        1 16.0 30.4 34.8 37.2
# 2 Quebec                   2 13.6 27.3 37.1 41.8
# 3 Quebec                   3 16.2 32.4 40.3 42.1
# 4 Québec    chilled        1 14.2 24.1 30.3 34.6
# 5 Québec                   2  9.3 27.3 35.0 38.8
# 6 Québec                   3 15.1 21.0 38.1 34.0
#                    X500  X675 X1000
# 1                  35.3  39.2  39.7
# 2                  40.6 41.4   44.3
# 3                  42.9  43.9  45.5
# 4 32.5 (umol/m^2 sec)  35,4  38.7
# 5                  38.6  37,5  42.4
# 6                 +38.9  39,6  41.4
```

Dimensions (rows & columns)

```r
dim(DF)
```

```
# [1] 13 10
```

```r
nrow(DF)
```

```
# [1] 13
```

```r
ncol(DF)
```

```
# [1] 10
```

Names of elements (columns)

```
names(DF)
```

```
#  [1] "Type"      "Treatment" "PlantNum"  "X95"
#  [5] "X175"      "X250"      "X350"      "X500"
#  [9] "X675"      "X1000"
```

```
colnames(DF)
```

```
#  [1] "Type"      "Treatment" "PlantNum"  "X95"
#  [5] "X175"      "X250"      "X350"      "X500"
#  [9] "X675"      "X1000"
```

```
rownames(DF)
```

```
#  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11"
# [12] "12" "13"
```

Look at a column

Remember: you can refer to elements within a data frame by *name*.

```
DF[, "Treatment"]
```

```
#  [1] "nonchilled" ""           ""           "chilled"
#  [5] ""           ""           "nonchilled" ""
#  [9] ""           "chilled"    ""           ""
# [13] ""
```

```
unique(DF$Type)
```

```
# [1] "Quebec"      "Québec"      "Mississippi"
```

!

Looks like there might be some missing values in the `Treatment` column, and inconsistencies in the `Type` column. We'll learn how to fix those soon, but these simple functions are already helping us understand our data.

str(): structure of an object

```
str(DF)
```

```
# 'data.frame':	13 obs. of  10 variables:
#  $ Type     : chr  "Quebec" "Quebec" "Quebec" "Québec" ...
#  $ Treatment: chr  "nonchilled" "" "" "chilled" ...
#  $ PlantNum : int  1 2 3 1 2 3 1 2 3 1 ...
#  $ X95      : num  16 13.6 16.2 14.2 9.3 15.1 10.6 12 11.3 10.5 .
#  $ X175     : num  30.4 27.3 32.4 24.1 27.3 21 19.2 22 19.4 14.9
#  $ X250     : num  34.8 37.1 40.3 30.3 35 38.1 26.2 30.6 25.8 18.
#  $ X350     : num  37.2 41.8 42.1 34.6 38.8 34 30 31.8 27.9 18.9
#  $ X500     : chr  "35.3" "40.6" "42.9" "32.5 (umol/m^2 sec)" ...
#  $ X675     : chr  "39.2" "41.4 " "43.9" "35,4" ...
#  $ X1000    : num  39.7 44.3 45.5 38.7 42.4 41.4 35.5 31.5 27.8 2
```

Tip

*The `str()` and `names()` functions can be used with **any** object.*

summary(): statistical summaries by column

```r
summary(DF)
```

```
#      Type            Treatment            PlantNum
#  Length:13          Length:13          Min.   :1
#  Class :character   Class :character   1st Qu.:1
#  Mode  :character   Mode  :character   Median :2
#                                        Mean   :2
#                                        3rd Qu.:3
#                                        Max.   :3
#
#       X95            X175            X250
#  Min.   : 7.7   Min.   :11.4   Min.   :12.3
#  1st Qu.:10.5   1st Qu.:18.0   1st Qu.:23.9
#  Median :11.3   Median :21.0   Median :30.4
#  Mean   :11.9   Mean   :21.4   Mean   :28.9
#  3rd Qu.:14.2   3rd Qu.:27.3   3rd Qu.:35.5
#  Max.   :16.2   Max.   :32.4   Max.   :40.3
#                                NA's   :1
#       X350          X500               X675
#  Min.   :13.0   Length:13          Length:13
```

Simple plots

```
plot(DF)
```

Spreadsheet-like `View()`

```
View(DF)
```

- This command opens a data frame in a spreadsheet-like view, which can be easier to navigate.
- In RStudio, you can achieve the same thing by clicking on an object name in the '*Environment*' pane (default upper-right)
  - The `View()` pane in RStudio (default upper-left; '*Source*') also allows for sorting and filtering, but these do not change the object in your session, only the view.

Encoding non-English characters

- If you are running R in Windows, you may notice that some values of the `Type` column look strange:

  "**QuÃ©bec**" instead of "Québec"

- There is nothing wrong with the file — this indicates a *mismatch* between the *encoding* used to write the file, and what `R` used to read it.

- Even though '`.csv`' files are plain text, letters (especially non-english characters) can be *encoded* in different ways to represent them in the computer.

- "`UTF-8`" is a character encoding standard designed to handle many non-english characters.

  - The example data file was written in "`UTF-8`"
  - Most OSes and many programs use "`UTF-8`" encoding by default.
  - But *Windows* uses "`latin1`" by default, and so does `R` ($<$ 4.2.0) when running in Windows.
  - Starting with v4.2.0, `R` uses "`UTF-8`" as the default encoding on Windows.

Read a csv file with a different encoding

- You can specify the encoding used in the file with the 'encoding' argument of `read.csv()`

```
DF <- read.csv(DF_path, skip = 2, encoding = "UTF-8")
```

- If reading a file that was created on a Windows computer and encoded in "latin1", on a different system (mac, Unix, linux, etc.) — or a recent version of R (>=4.2) on Windows — you can specify that, too:

```
read.csv(DF_path, skip = 2, encoding = "latin1")
```

Microsoft Excel on Windows

Excel in Windows is capable of saving files in `.csv` format with "UTF-8" encoding, but it adds extra contents to the file (a "BOM") that makes it difficult to read with `read.csv()`.
See the "Extras" document from this workshop for details on how to deal with that.
Or use the `readr` package (coming up) and don't worry about it. :)

Know Your Data

- These functions are useful for exploring different aspects of a loaded data set
- But they won't tell you if these are *correct*.
- Ideally, you should always "Know Your Data", and use these functions to verify that the data was loaded correctly.
  - Are the number of rows and columns what you expected?
  - Are the different columns of the expected type (numeric, character, etc.)?
  - Are the values in the expected range and format?
  - Is anything missing, or different than expected?

The `CO2` dataset: background

The example data file is based on the `'CO2'` dataset available in `R` (`?CO2`), with a few changes added to make things interesting.

From the documentation:

> *The CO2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.*

Exercise 1: what's wrong with this data?

The original dataset has the following properties (`str(CO2)`):

- 84 rows and 5 columns

Column Name	Description
`Plant`	factor with 12 levels: Qn1, Qn2, … Mc3, Mc1
`Type`	factor with 2 levels: "Quebec" and "Mississippi"
`Treatment`	factor with 2 levels: "nonchilled" and "chilled"
`conc`	numeric: ambient carbon dioxide concentrations (mL/L)
`uptake`	numeric: carbon dioxide uptake rates (μmol/m^2·sec)

Your turn

Using the functions described in this section, can you identify some possible issues and differences with the data set you loaded?
Spoiler alert: *suggested answers on the next slide.*

Exercise 1: what *is* wrong with this data

- The data we loaded has different dimensions!
  - Values from the `conc` column are shown as *column names*
  - `uptake` values are the values of these columns
  - This isn't necessarily *bad*: such a structure can be useful for presentation and interpretation by people, but it is not *tidy* and less convenient for analysis & visualization (more on this later).
- Some of the `uptake` values are *character*, but should be *numeric*
- One of the `Type` values is spelled inconsistently: "Quebec"/"Québec"
- Some values in the `Treatment` column are empty
  - The value is only included when it changes
- The `PlantNum` column does not contain a unique identifier, as in the original `Plant` column
  - The values are no longer *unique*, without also considering the `Type` and `Treatment` columns.

There are other differences you may have noticed: we'll look at ways to identify these automatically later.

Section 5

Re-using your code: scripts and other files

Re-using code

Before we practice cleaning our data, and saving it to use later ...

- Imagine having to repeat the multiple steps to *load*, *clean*, and *save* a dataset.
  - How will you remember which *packages* you had to load?
  - How will you remember *all* the steps, and their order?
  - What if you need to change *one* step, but repeat all preceding steps?
  - How can you share your code with others, so that they can check your work, or replicate your results?
  - How will you write out complex operations that require multiple steps, repeated operations, or only do things under certain conditions?
- The answer to all of these questions is: a ***script***

Scripts and related files will also make it easier for you to follow along with examples as they get more complicated—copy & paste into the console less often!

Scripts

An `R` *script* is a file that stores `R` code in *plain text*

- They have a `.R` file extension
- They are plain text files
  - so any text editor can read & write them
  - they also work well with **v**ersion **c**ontrol **s**ystems, like `git`, GitHub, and GitLab)
- All the code in a script can be run in order
  - i.e., a *program*
- They make it easy to re-use code
- Scripts provide a record of the steps in a program or analysis
  - results are more *reproducible*
  - the code is a form of documentation

Make a new script

- In your `R` interface (R GUI, RStudio, IDE, etc.), open a new R script

Application	Menu item	Keyboard shortcut (mac shortcut)
R GUI	`File > New Document`	`CTL+N (CMD+N)`
RStudio	`File > New File > R Script` 	`Shift+CTL+N (Shift+CMD+N)`

- Save it with a name like "`my_first_script.R`"
  - You can save it in the same location as the slides for this workshop (i.e., the folder *containing* the '`data`' folder.)

Add some code to your script

- Paste in the following code to your script file:

```
DF_path <- file.path("data", "data_example.csv")
file.exists(DF_path)

DF <- read.csv(DF_path, skip = 2, encoding = "UTF-8")

colnames(DF)
plot(DF)
```

- and save it.

Run R code in scripts

Most IDEs have a shortcut to send portions of R code (a line or *statement* spanning multiple lines) to an R session:

- R GUI: `CTL+Return` (mac: `CMD+Return`)
- RStudio: `CTL+Return` (mac: `CMD+Return`)

You can run *all* the code in a script in different ways:

- The `source()` function, with a path to the script file as an argument
  - The code will run in the current session.
  - `?source`
  - `source("my_script.R")`
- Run R in "*batch mode*"
  - "*batch mode*" is **not** interactive (no prompt)
  - It is usually invoked from a terminal or other command-line (outside an R GUI)
  - The code in the script will run in a new session
  - You can capture output in a separate file
  - `?BATCH`

Comments

- The '#' character denotes a *comment* in R
  - *Everything on a line* after a comment character is ignored by R
  - There are no 'multi-line' comments in R

```r
print("this is R code")  # this is a comment
```

- You can make an entire line a comment by putting a comment character at the beginning.
  - Divide your code into *sections*
    `Shift+CTL+R` (`Shift+CMD+R`) in RStudio

```r
# SECTION ---------------------------------------------------
```

  - Create 'comment headers' for your scripts:

```r
##############################################################
### Title
### Project or description
### Author Name            R vX.X.X              YYYY-MM-DD
##############################################################
```

Comments in code

- You can put a comment beside a line of code
  (even in the middle of a mult-line statement)
    - R will ignore the rest of the line, and continue reading code on the next line

```
DF <-           # short for "data frame"
  read.csv(     # read a csv file
    DF_path,    # path to file
    skip = 2    # skip lines at top of file (not data)
  )
```

- Use comments to
    - organize your code (divide it into sections)
    - explain the code, where relevant
    - "comment-out" code temporarily, to stop it from running without deleting it (useful for debugging).

      `Shift+CTL+C` (`Shift+CMD+C`) comment a line in RStudio

Open a script

- All the code shown in the slides for this workshop has been collected in a script file: "`R_data_scripting.R`" (in the '`source`' folder)
- Open it to follow along for the rest of the workshop.

Application	Menu item	Keyboard shortcut (mac shortcut)
R GUI	`File > Open Document...`	`CTL+O (CMD+O)`
RStudio	`File > Open File...`	`CTL+O (CMD+O)`
		

Set the Working Directory to source file location in **RStudio**

- Menu item: "`Session > Set Working Directory > To Source File Location`"
- This makes it easy to use *relative paths* in your script, relative to the location of the script file itself.

For this workshop

- All the code in this document assumes that the working directory is the *same directory* as where the script file is.

Section 6

The `tidyverse` collection of packages

The `tidyverse`

```r
install.packages("tidyverse")
help(package="tidyverse")
```

- The `tidyverse` is an "opinionated" collection of packages that are designed to work together.
- All packages share an underlying design philosophy, grammar, and data structures.
  - *Unlike base R*
  - Shared naming conventions (e.g., '`_`' instead of '`.`' in function names)
  - Emphasis on functions that do one thing well
  - Designed to be combined together to achieve complex operations
- `tidyverse` is under active development.
  - New functions and features sometimes replace or supersede old ones.
  - No guarantee that functions will continue to work the same way in future versions.

Core `tidyverse` packages

Today, we will focus on a few of the core `tidyverse` packages for loading, cleaning, and manipulating data:

- readr, readxl for **loading** data
- dplyr for **manipulating** data (values)
- tidyr for **reshaping** data
- stringr for working with **strings**

Section 7

Load Data: The `readr` & `readxl` Packages

The readr package: reading data

readr		base R	
read_csv()	comma separated values	read.csv()	
read_csv2()	';' as delimiter (allows ',' for decimals)	read.csv2()	',' for decimals, ';' as separator
read_tsv()	tab separated values	read.delim()	delimited files (tab is default)
read_delim()	(generic) files with any delimiter	read.table()	
read_fwf()	fixed width files	read.fwf()	

readr descriptions based on #dsbox

Read a csv file using `read_csv()`

- In keeping with Tidyverse conventions, functions are names with words separated by "_"
  - instead of "." or `camelCase`, as in many base `R` functions

```
library(readr)
DF_readr <- read_csv(DF_path, skip = 2)
```

```
# Rows: 13 Columns: 10
# -- Column specification ------------------------------------
# Delimiter: ","
# chr (3): Type, Treatment, 500
# dbl (6): PlantNum, 95, 175, 250, 350, 1000
# num (1): 675
#
# i Use `spec()` to retrieve the full column specification for this
# i Specify the column types or set `show_col_types = FALSE` to quie
```

Exercise 2: compare results from `read.csv()` and `read_csv()`

- Use the functions we learned earlier to inspect and compare the results of `read.csv()` (in base `R`) and `read_csv()` (from the `readr` package)
- There's a script file in the `exercises` folder to get you started.
  - "`R2_exercise_2.R`"

Spoiler alert:

suggested answers on the next slide.

Exercise 2: comparison of `read.csv()` and `read_csv()`

- The column names are different
  - `read.csv()` automatically applies `make.names()` to the column names to make 'syntactically valid' names to use in R.
  - convenient, but not always what we want.
  - there are other 'cleaning' functions available (e.g., `clean_names()` in the `janitor` package)
- `read_csv()` automatically replaced empty strings in the `Treatment` column with `NA`s.
- `read_csv()` left the '675' column as numeric, but ignored the commas, resulting in larger numbers.
- `read_csv()` produces a `tbl_df` (*tibble*) object, not a simple `data.frame`

Tibbles: `data.frames` reimagined

- A 'tibble' (`class() == "tbl_df"`) is "a modern reimagining of the data.frame".
  - See the package documentation for details.
- Many tidyverse functions produce tibbles by default.
- **Tibbles are *also* `data.frames`**, and *inherit* from that class.
  - functions that work with `data.frames` should also work with tibbles.
  - but some may behave differently (by design).
  - for example, `print()`ing a tibble includes slightly more information, and only prints a few rows and columns by default, preventing large datasets from overwhelming your console.
  - when indexing a tibble, it will *not* do partial matching on column names, making it clear if a column exists or not
- For most purposes, tibbles are interchangeable with `data.frames`.
  - A tibble can be converted to a 'plain' `data.frame` with `as.data.frame()` if necessary.

Tibble examples

```
print(DF_readr, n=2)
```

```
# # A tibble: 13 x 10
#   Type   Treatment  PlantNum  `95` `175` `250` `350` `500`
#   <chr>  <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
# 1 Quebec nonchilled        1  16    30.4  34.8  37.2 35.3
# 2 Quebec <NA>              2  13.6  27.3  37.1  41.8 40.6
# # i 11 more rows
# # i 2 more variables: `675` <dbl>, `1000` <dbl>
```

```
is.null(DF$Treat)
```

```
# [1] FALSE
```

```
is.null(DF_readr$Treat)
```

```
# Warning: Unknown or uninitialised column: `Treat`.
# [1] TRUE
```

`read_csv()`: column names

- The default for `read_csv()` is to ensure all column names are *unique*, but not necessarily *syntactically valid*
- You can still refer to columns with syntactically 'invalid' names:
  - use functions that allow names as characters,
  - quote names with backticks (`` ` ``)

```
DF_readr[, "95"]  # still a `data.frame` (with 1 column)
DF_readr[["95"]]  # vector
DF_readr$`95`     # quoted name
```

read_csv(): treat column names

- Change how `read_csv()` treats column names with the 'name_repair' argument

```
read_csv(
  DF_path, skip = 2,
  name_repair = "universal" # make names unique and syntactic
)

read_csv(
  DF_path, skip = 2,
  name_repair = make.names  # a function: same as read.csv()
)
```

read_csv(): guessing column types

- By default, `read_csv()` prints a message summarizing what it did, including *guessing the data type* of each column.
  - *.csv files do not include this information as metadata*
- Control how columns are guessed with the guess_max argument:

```
# use the first 2 rows to guess column types (less successful,
  read_csv(DF_path, skip = 2, guess_max = 2)
```

```
# Warning: One or more parsing issues, call `problems()` on your dat
# frame for details, e.g.:
#   dat <- vroom(...)
#   problems(dat)
```

```
# use *all* rows to guess column types
# - slow: has to read *every row* twice.
  read_csv(DF_path, skip = 2, guess_max = Inf)
```

read_csv(): specify column types

- If you know what the column types are (or should be), you can tell `read_csv()` what they are with the `col_types` argument.
  - for large datasets, this can be faster: read rows once
  - avoid bad guesses.

```
## Specify column types with a compact string
read_csv(DF_path, skip = 2, col_types = "cccdddddd")

## Or use a `column specification`
# extract specification from tibble
col_spec <- spec(DF_readr)
# change a column to numeric (double)
col_spec$cols[["500"]] <- col_double()
read_csv(DF_path, skip = 2, col_types = col_spec)

# ?read_csv for more options
```

`read_csv()`: all columns as strings

- In extreme cases, you can read everything as '`character`', then clean and coerce to other data types within `R`

```
# read all columns as character
read_csv(DF_path, skip = 2,
         col_types = cols(.default = col_character())
         )
```

```
# # A tibble: 13 x 10
#    Type      Treatment PlantNum `95`  `175` `250` `350` `500`
#    <chr>     <chr>     <chr>    <chr> <chr> <chr> <chr> <chr>
#  1 Quebec    nonchill~ 1        16    30.4  34.8  37.2  35.3
#  2 Quebec    <NA>      2        13.6  27.3  37.1  41.8  40.6
#  3 Quebec    <NA>      3        16.2  32.4  40.3  42.1  42.9
#  4 Québec    chilled   1        14.2  24.1  30.3  34.6  32.5~
#  5 Québec    <NA>      2        9.3   27.3  35    38.8  38.6
#  6 Québec    <NA>      3        15.1  21    38.1  34    +38.9
#  7 Mississ~ nonchill~ 1        10.6  19.2  26.2  30    30.9
#  8 Mississ~ <NA>      2        12    22    30.6  31.8  32.4
#  9 Mississ~ <NA>      3        11.3  19.4  25.8  27.9  28.5
```

read_csv(): missing values

- Use the na argument to supply a list of values to replace with NA.
  - This is applied to **all** columns.

```
read_csv(DF_path, skip = 2,
         na = c(".", "NA")  # will not replace empty strings
         )
```

```
# # A tibble: 13 x 10
#    Type     Treatment PlantNum  `95` `175` `250` `350` `500`
#    <chr>    <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
#  1 Quebec   "nonchil~        1  16    30.4  34.8  37.2 35.3
#  2 Quebec   ""               2  13.6  27.3  37.1  41.8 40.6
#  3 Quebec   ""               3  16.2  32.4  40.3  42.1 42.9
#  4 Québec   "chilled"        1  14.2  24.1  30.3  34.6 32.5~
#  5 Québec   ""               2   9.3  27.3  35    38.8 38.6
#  6 Québec   ""               3  15.1  21    38.1  34   +38.9
#  7 Mississ~ "nonchil~        1  10.6  19.2  26.2  30   30.9
#  8 Mississ~ ""               2  12    22    30.6  31.8 32.4
#  9 Mississ~ ""               3  11.3  19.4  25.8  27.9 28.5
# 10 Mississ~ "chilled"        1  10.5  14.9  18.1  18.9 19.5
```

The `readxl` package

Provides functions for reading from (but not writing to) Microsoft Excel files (`.xls` and `.xlsx`)

```
library(readxl)    # load the package
## Documentation: ?read_excel  help(package="readxl")
## use an example included in the package
xl_path <- readxl_example("datasets.xlsx")
excel_sheets(xl_path)  # get the names of the sheets
```

```
# [1] "iris"     "mtcars"   "chickwts" "quakes"
```

```
## read a specified sheet from the Excel file
iris_xl <- read_excel(xl_path, "iris")
```

Example 3: read a messy Excel file

- `read_excel()` has many of the same arguments as `read_csv()` to control how data is imported.
- Use the script file in the "`examples`" folder as a starting point:
  - "`R2_exercise_3.R`"

```
XL_path <- readxl_example("deaths.xlsx")
XL <- read_excel(XL_path, ...)
```

Example 3: read a messy Excel file

- `read_excel()` has many of the same arguments as `read_csv()` to control how data is imported.
- Use the script file in the "`examples`" folder as a starting point:
    - "`R2_exercise_3.R`"

```
XL_path <- readxl_example("deaths.xlsx")
XL <- read_excel(XL_path, ...)
```

Tip

Use the '`range`' argument to read data from a specific range in a sheet, ignoring contents outside this range (rows above & below, columns before & after)

Section 8

Manipulate Data: The `dplyr` Package

dplyr: a grammar of data manipulation

- `dplyr` provides many functions that follow a coherent framework or "*grammar*"
- They are intended to help you focus on *what* you want to do, and translate your thoughts into code.
- High-level functions have active names and called "**verbs**" — they describe what they do.
- `dplyr` and `tidyselect` provide many "**helper functions**" that work *inside* verbs and other `tidyverse` functions to make common tasks easier to translate into code.
    - These functions may not work on their own, outside of `dplyr` verbs and `tidyr` functions (see `?"faq-selection-context"`).

dplyr verbs

Verbs can be grouped based on the component of the dataset that they work with[2]:

- Columns:
  - `select()` changes whether or not a column is included.
  - `relocate()` changes the order of the columns.
  - `rename()` changes the name of columns.
  - `mutate()` changes the *values* of columns and creates new columns.
- Rows:
  - `filter()` chooses rows based on column values.
  - `slice()` chooses rows based on location.
  - `arrange()` changes the order of the rows.
- Groups of rows:
  - `group_by()` defines groups of rows.
  - `summarise()` collapses a group into a single row.

[2] https://dplyr.tidyverse.org/articles/dplyr.html#single-table-verbs

Load dplyr

- When you load `dplyr`, you may see a message saying "objects are masked":

```
library(dplyr)
```

```
#
# Attaching package: 'dplyr'

# The following objects are masked from 'package:stats':
#
#     filter, lag

# The following objects are masked from 'package:base':
#
#     intersect, setdiff, setequal, union
```

- Some functions in `dplyr` have the same name as functions in other packages! (`base`, `stats`)

`package::object` notation

- When an "object is masked", it means the new one (`dplyr` function) will take precedence over the one being "masked" when a function of that name is called.

```
?filter  # more than one result!
```

- To avoid confusion, you can specify which package you mean with the "`package::object`" notation (*functions* are a class of *object*):

```
?filter  # more than one result
?dplyr::filter
?stats::filter
```

- Because `dplyr` has *masked* the names from the other packages, we can simply call these functions normally (e.g., `filter()`), but if we wanted to use a masked function, we would have to use the special notation: `stats::filter()`
  - We won't be using the masked functions today, however.

Section 9

`dplyr` Verbs: `select()` & `filter()` Parts of a Data Frame

Columns: `select()`

- Select (and rename) columns in a data frame, using a concise mini-language (`<tidy-select>`)
- Select by name: ‘bare names’, like regular variables; or character names

```
select(DF, Type, Treatment, PlantNum)
select(DF, "Type", "Treatment", "X95")
select(DF, Type:PlantNum)
```

- Select by position:

```
select(DF, 2:5)
```

- or both, while changing names and order along the way:

```
select(DF, c(Type, 4:6, Plant = PlantNum))
```

Columns: selection helpers

- Various **helper functions** (*selection helpers*) add ways to `select()` columns *based on criteria* (dynamically).
- Some examples (see `?select` for more):

```
select(DF, starts_with("X"))
select(DF, !starts_with("X"))
select(DF, contains("m"))

select(DF, where(is.character) & starts_with("X"))

select(DF, any_of(c("Type", "Treatment", "95")))
```

Columns: selection helpers

- Various **helper functions** (*selection helpers*) add ways to `select()` columns *based on criteria* (dynamically).
- Some examples (see `?select` for more):

```
select(DF, starts_with("X"))
select(DF, !starts_with("X"))
select(DF, contains("m"))

select(DF, where(is.character) & starts_with("X"))

select(DF, any_of(c("Type", "Treatment", "95")))
```

- `starts_with()` & `contains()` match *patterns* in names
- `where()` applies a function (`is.character()`, in this case) to each column: those where the function returns `TRUE` are kept.
- `any_of()` matches names in a character vector; `all_of()` is similar, but names that don't exist cause an *error*.

Rows: filter()

- `filter()` retains rows that satisfy *all* the conditions specified
  - expressions must return a vector of *logical* values (`TRUE` or `FALSE`)

```
filter(DF, X95 < 10)
filter(CO2, conc == 95, uptake < 10)
filter(CO2, conc == 95 | uptake < 10) # | == "OR" operator

filter(DF, Treatment != "")
filter(DF, !Type %in% c("Quebec", "Mississippi"))
filter(DF, !Type %in% unique(CO2$Type))

filter(DF, X175 > mean(X175))
```

`select()` & `filter()` work well together

- For example, you can filter on a column, then remove it with `select()`

```
select( filter(DF, Treatment == "chilled"),
        where(is.numeric)
)
```

```
#   PlantNum  X95 X175 X250 X350 X1000
# 1        1 14.2 24.1 30.3 34.6  38.7
# 2        1 10.5 14.9 18.1 18.9  21.9
```

- But this is clunky! This is easier to follow:

```
DF %>% filter(Treatment == "chilled") %>%
  select(where(is.numeric))
```

```
#   PlantNum  X95 X175 X250 X350 X1000
# 1        1 14.2 24.1 30.3 34.6  38.7
# 2        1 10.5 14.9 18.1 18.9  21.9
```

A ‘pipe’ operator

- `%>%` allows you to pass results from an expression on the left-hand side (LHS) as an argument (usually the first) to a *function call* on the right-hand side (RHS).
  - you can read a pipe operator as “then” (`a %>% b()` = “*a then b*”)

This expression …	is equivalent to:
`x %>% f()`	`f(x)`
`x %>% f(y)`	`f(x, y)`
`x %>% f(y, z = .)`	`f(y, z = x)`
`x %>% f %>% g %>% h`	`h(g(f(x)))`

- This can make code easier to read, as expressions are written and evaluated from *left to right*, rather than from *inside to outside* nested parentheses.

magrittr's 'forward-pipe' operator

Figure 1: "La Trahison des Images" ("The Treachery of Images") or "Ceci n'est pas une pipe" ("This is not a pipe") by René Magritte.

- The magrittr package (included with tidyverse) provides a "forward-pipe operator":

```
%>%      # ?magrittr::`%>%`
```

- The `magrittr` package is automatically loaded when loading most `tidyverse` packages (e.g., `tidyr`, `dplyr`, `ggplot2`).
  - These packages were designed to work with this operator, and use it themselves.
  - It is often unnecessary to load `magrittr` separately, unless you are **not** using these other packages.

R now has a 'native' pipe operator

- A pipe operator was introduced in base R in v4.1 (May 2021)[3]:

```
|>     # ?pipeOp
```

- It was inspired by the "forward pipe operator" introduced by `magrittr`, but is more streamlined. See these links for details:
  - Differences between the base R and magrittr pipes
  - "Understanding the native R pipe |>"
- Because '`|>`' is new, many examples online still use `magrittr`'s '`%>%`'.
- But '`|>`' is always available *in R >= v4.1*, without having to load additional packages.
- This document will use '`%>%`' in the examples, for consistency and because many `tidyverse` functions were designed to work with it.

[3] https://cran.r-project.org/bin/windows/base/old/4.1.0/NEWS.R-4.1.0.html

Pipe: Exercise

Insert a pipe in R Studio with `CTL+Shift+M` (`CMD+Shift+M`)

Re-write the expressions using pipes:

```
# (1)
sum(1:10)

# (2)
filter(CO2, conc < 100)

# (3)
filter(
  select( DF,
    where(is.character)
  ),
  Treatment == "")

# (4)
gsub("X", "", names(DF))
```

Pipe: Exercise

Insert a pipe in R Studio with `CTL+Shift+M` (`CMD+Shift+M`)

Re-write the expressions using pipes:

```r
# (1)
sum(1:10)

# (2)
filter(CO2, conc < 100)

# (3)
filter(
  select( DF,
    where(is.character)
  ),
  Treatment == "")

# (4)
gsub("X", "", names(DF))
```

```r
# (1)
1:10 %>% sum()

# (2)
CO2 %>% filter(conc < 100)

# (3)
DF %>%
  select(
    where(is.character)
  ) %>%
  filter(Treatment == "")

# (4)
DF %>% names() %>%
gsub("X", "", .)
```

`dplyr` conventions

All `dplyr` verbs (and other related `tidyverse` functions) share a few things in common:

- The first argument is a data frame (or tibble).
- Other arguments describe what to do with the data frame.
- You can refer to columns in the data frame directly without using `$`.
- The result is a new data frame.

These features work well with the pipe operator, and can help build clear and efficient workflows.

Because all the functions have these in common, it makes it easier to extend your understanding to new functions.

Find problematic values systematically

- Find columns that are character, but we expect to be numeric:

```
cols_charn <- DF %>%
  select(starts_with("X") & where(is.character)) %>%
  names()
```

- Loop through the identified column names (using `for`) and print values that would be `NA` if converted to numeric
  - `suppressWarnings()` suppresses the warnings we expect from `as.numeric()` in this case.

```
for (col in cols_charn) {
  DF %>% select(1:3, all_of(col)) %>%
    filter( get(col) %>% as.numeric() %>% is.na() %>%
              suppressWarnings() ) %>%
    print()
}
```

```
#     Type Treatment PlantNum                     X500
# 1 Québec   chilled        1 32.5 (umol/m^2 sec)
#     Type Treatment PlantNum X675
# 1 Québec   chilled        1 35,4
# 2 Québec                  2 37,5
# 3 Québec                  3 39,6
```

Exercise

User-defined function

- Define your own functions with the `function` function!
  - function code goes between braces: {}
  - specify what value is returned with the `return()` function

```
'%==%' <- function (v1, v2) {
  same <- (v1 == v2) | (is.na(v1) & is.na(v2))
  same[is.na(same)] <- FALSE
  return(same)                    # return the result
}

# test it:
c(1, NA, 3, 4 , NaN) %==% c(1, NA, 1, NA, NaN)
```

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

```
c(1, NA, 3, 4 , NaN)  ==  c(1, NA, 1, NA, NaN)
```

```
# [1]  TRUE    NA FALSE    NA    NA
```

- This code defines a function that compares two vectors, accounting for missing values (`NA`)

An infix operator

```
c(1, NA, 3, 4 , NaN) %==% c(1, NA, 1, NA, NaN)
```

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

- This function is also a special type called an "*infix operator*", which goes between two objects (it's arguments) like an operator, instead of a 'typical' function call
  - it has exactly **2** arguments (lhs, rhs)
  - the name begins *and* ends with a percent symbol (%)

Section 10

`dplyr` Verbs: Modify Data Columns

Modify Columns: `mutate()`

- `mutate()` creates new columns, or modifies existing ones, as functions of existing columns.
  - it is a `dplyr` workhorse, used for many tasks, since it allows you to modify values *systematically*.

Add columns:

```
DF %>% mutate(
  Trt_n = Treatment %>% nchar(),
  Xsum = X95 + X175
  )
```

Modify columns:

```
DF %>% mutate(X95 = X95 / mean(X95))
```

`mutate()` helper functions

- refer to values in the previous / next row with `lag()` and `lead`

```
DF %>% mutate(
  Plant_lag  =  lag(PlantNum),
  Plant_lead = lead(PlantNum, n=2)
)
```

- refer to row numbers with `row_number()`

```
DF %>% mutate(
  Plant_row = PlantNum < row_number()
)
```

mutate(): conditional values

- case_when() lets you apply multiple if/else statements to a vector of values.
  - conditions go first
  - the value returned if the condition is true goes after a tilde (~)
  - multiple statements are separated by commas
  - If no default is specified (.default =), anything that does not match any condition is replaced with NA

```
DF %>% mutate(
  Type_ab = case_when(
    Type == "Quebec"      ~ "QC",
    Type == "Mississippi" ~ "MS",
    .default = as.character(Type)
  )
)
```

`dplyr` *semantics*

`dplyr` verbs and helper functions let you refer to column names of the data frame directly in their arguments as regular variables — without having to quote them as strings. But these names have different meanings (semantics) in different verbs.

- **"select semantics"** (`<tidy-select>`): in `select()` and similar functions, a column name refers to its *position* in the data frame.
  - you can refer to a column as a quoted string in `select()`, and it is interpreted as a reference to the column.
- **"mutate semantics"** (`<data-masking>`): in `mutate()` and similar functions (`group_by()`, `summarise()`, `filter()`, etc.), a column name refers to a *vector of values*.
  - you cannot supply a column name as a string in `mutate()`, because it is treated as a vector of length 1, rather than a reference to a column of values.
- Helper functions only work in one context or the other, so knowing the difference will tell you which helper functions to use when.

Exercise

Section 11

Clean Some Data With the `stringr` Package

Cleaning some columns in the example data

- Normalize values of the 'Type' column:
    - using `if_else()`

```
DF_clean1_type <- DF %>%
  mutate(
    Type = if_else(Type == "Mississippi", Type, "Quebec")
  )
```

Cleaning some columns in the example data

- Normalize values of the 'Type' column:
  - using `if_else()`

```
DF_clean1_type <- DF %>%
  mutate(
    Type = if_else(Type == "Mississippi", Type, "Quebec")
  )
```

- could also use `case_when()`

```
DF %>%
  mutate(
    Type = case_when(
      Type == "Québec"  ~ "Quebec",
      .default = Type
    )
  )
```

Convert character column to numeric (replace characters)

- To clean character columns, we will use functions from the `stringr` library, which provides many useful functions for working with, and manipulating *strings*.

```
library(stringr)
# help(package="stringr")
# vignette("stringr")
```

- Replace ',' with '.' in `X675` column, and convert to numeric:

```
DF_clean2_675 <- DF_clean1_type %>%
  mutate(
    # Replace "," with "."
    X675 = str_replace(X675, ",", "."),
    # convert to numeric
    X675 = as.numeric(X675)
  )
```

Convert character column to numeric (drop characters)

- Remove non-numeric characters from `X500` column, and convert to numeric:
  - We can use `str_split_i()` from the `stringr` package to 'split' each string into pieces separated by spaces (' '), and keep only the first piece (`i=1`).
  - i.e., drop all text (in each row) after the first space

```
DF_clean3_500 <- DF_clean2_675 %>%
  mutate(
    # drop everything after the first space:
    X500 = str_split_i(X500, " ", i=1),
    # convert to numeric
    X500 = as.numeric(X500)
  )
```

Clean ‘Treatment’ column

- The ‘`Treatment`’ column has some empty values
- A value is only present when it changes
- We can use the `fill()` function from the `tidyr` package
  - without loading the package, by specifying the function with `package::object` notation

```
DF_clean4_cols <- DF_clean3_500 %>%
  # replace empty strings with NA
  mutate(Treatment = na_if(Treatment, "")) %>%
  # Fill Down to replace NAs (tidyr)
  tidyr::fill(Treatment, .direction = "down")
```

Exercise

Section 12

dplyr Verbs: Grouped Data

Define groups: `group_by()`

- Group rows based on combinations of column values

```
DF %>% group_by(Type, PlantNum)  # no visible change
```

- dplyr verbs are applied to each group of rows

```
DF %>% group_by(Type, PlantNum) %>%
  filter(row_number() == 1)
DF %>% group_by(Type, PlantNum) %>%
  arrange(Type, PlantNum) %>%
  mutate(Norm95 = X95 / mean(X95))
```

- Grouping columns are excluded from the operations

```
DF %>% group_by(Type, PlantNum) %>%
  select(starts_with("X"))
```

```
# Adding missing grouping variables: `Type`, `PlantNum`
```

Collapse groups: `summarise()` / `summarize()`

- Without groups specified, `summarise()` treats the data frame as a single group

```
DF_clean4_cols %>% summarise(n(), mean(X95))
```

```
#   n() mean(X95)
# 1  13     11.91
```

- But `summarise()` is most useful when applied to grouped data

```
DF_clean4_cols %>% group_by(Type, PlantNum) %>%
  summarise(n = n(), sum(X95))
```

```
# `summarise()` has grouped output by 'Type'. You can
# override using the `.groups` argument.
```

- `summarise()` automatically drops the last level from the groups.
- `summarise()` & `summarise()` are synonyms (same function).

summarise() multiple columns with across()

- summarise() uses *mutate semantics*, but across() applies a **function** to multiple columns, specified using *select semantics*
  - i.e., across() lets you use *select semantics* in a context where you would normally use *mutate semantics*
- A function can be specified by name

```
DF_clean4_cols %>%
  summarise( across(where(is.numeric), max) )
```

```
#   PlantNum  X95 X175 X250 X350 X500 X675 X1000
# 1        3 16.2 32.4   NA 42.1 42.9 43.9    NA
```

```
DF_clean4_cols %>% group_by(Treatment) %>%
  summarise( across(starts_with("X"), max) )
```

```
# # A tibble: 2 x 8
#   Treatment    X95  X175  X250  X350  X500  X675 X1000
#   <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
# 1 chilled     15.1  27.3  NA    38.8  38.9  39.6  NA
# 2 nonchilled  16.2  32.4  40.3  42.1  42.9  43.9  45.5
```

summarise() across() with ad hoc function

- For more complex operations, you may have to define a *custom function* or define an *ad hoc function* to include additional arguments

```
DF_clean4_cols %>%
  summarise(
    across(where(is.numeric),
           function(x) max(x, na.rm = TRUE)
    )
  )
```

```
#   PlantNum  X95 X175 X250 X350 X500 X675 X1000
# 1        3 16.2 32.4 40.3 42.1 42.9 43.9  45.5
```

summarise() across() with "lambda notation"

- You can also define ad hoc functions using a special "lambda" notation
  - refer to the value in the column with '.x'

```
DF_clean4_cols %>%
  summarise( across(everything(), ~ sum(is.na(.x))) )
```

```
#   Type Treatment PlantNum X95 X175 X250 X350 X500 X675
# 1    0         0        0   0    0    1    0    0    0
#   X1000
# 1     1
```

Tip

`sum(is.na())` *is a great way to count the number of missing values in a column.*

Locate duplicate rows in example data

- We can combine dplyr verbs, like `summarize` and `filter` to quickly locate issues

```
DF_clean4_cols %>%
  group_by(Type, Treatment, PlantNum) %>%
  summarise(n = n(), .groups = "drop") %>%
  filter(n > 1)
```

```
# # A tibble: 1 x 4
#   Type        Treatment PlantNum     n
#   <chr>       <chr>        <int> <int>
# 1 Mississippi chilled          2     2
```

Inspect duplicate rows in example data

- Luckily in this case, the missing values in the duplicate rows are *not* missing in the other row

```
DF_duprows <- DF_clean4_cols %>%
  group_by(Type, Treatment, PlantNum) %>%
  filter(n() > 1)
DF_duprows %>%
  mutate(across(everything(), ~ near(.x, max(.x, na.rm = TRUE)
```

```
# # A tibble: 2 x 10
# # Groups:   Type, Treatment, PlantNum [1]
#   Type      Treatment PlantNum X95   X175  X250  X350  X500
#   <chr>     <chr>        <int> <lgl> <lgl> <lgl> <lgl> <lgl>
# 1 Mississi~ chilled          2 TRUE  TRUE  NA    TRUE  TRUE
# 2 Mississi~ chilled          2 TRUE  TRUE  TRUE  TRUE  TRUE
# # i 2 more variables: X675 <lgl>, X1000 <lgl>
```

Combine duplicate rows in example data

- We can therefore collapse the duplicate rows using `summarise()`

```
DF_clean5_rows <- DF_clean4_cols %>%
  group_by(Type, Treatment, PlantNum) %>%
  summarise(
    across(where(is.numeric), ~ max(.x, na.rm = TRUE)),
    .groups = "drop"
  ) %>%
  ## Re-sort to original order
  arrange(desc(Type), desc(Treatment), PlantNum)
```

Exercise

Section 13

Reshape Data: The `tidyr` Package

Tidy data

> *"Happy families are all alike; every unhappy family is unhappy in its own way."*
> *— Leo Tolstoy*

> *"Tidy datasets are all alike but every messy dataset is messy in its own way."*
> *— Hadley Wickham (doi: 10.18637/jss.v059.i10)*

- Tidy datasets provide a standardized way to link the *structure* of a dataset (its physical layout) with its *semantics* (its meaning).
- A tidy dataset follows three interrelated rules:
  - Each variable must have its own column.
  - Each observation must have its own row.
  - Each value must have its own cell.

The `tidyr` package

```
library(tidyr)
```

The `tidyr` package provides tools for making data *tidy*:

- "**Pivoting**" converts between *long* and *wide* forms.
  - `pivot_longer()` and `pivot_wider()`
- "Rectangling" turns deeply nested lists (e.g., JSON) into tidy tibbles.
  - *Not covered here: see the vignette.*
- "Nesting" converts grouped data into nested data frames.
  - *Not covered here: see the vignette.*
- **Splitting** and **combining** character columns.
  - `separate_...()` a single character column into multiple columns
  - `unite()` multiple columns into a single character column.
- Handling **missing values**
  - make implicit missing values explicit with `complete()`
  - make explicit missing values implicit with `drop_na()`
  - replace missing values with a known value using `replace_na()` or `fill()` them with the next/previous value

Re-create the 'Plant' column

- Our example data is still missing unique values of 'Plant', made up of the first character of the 'Type' and 'Treatment' columns, and a unique number.

```
CO2 %>% pull(Plant) %>% unique() %>% as.character()
```

```
#  [1] "Qn1" "Qn2" "Qn3" "Qc1" "Qc2" "Qc3" "Mn1" "Mn2" "Mn3"
# [10] "Mc1" "Mc2" "Mc3"
```

- We can create temporary columns with `mutate()`

```
DF_clean5_rows %>% select(Type, Treatment) %>%
  ## Create columns with the first letter of each row
  mutate(
    Type.tmp      = str_sub(Type, 1, 1),
    Treatment.tmp = str_sub(Treatment, 1, 1),
  )
```

Combine columns

- We can create temporary columns with **mutate**(), then combine them with **unite**()

```
DF_clean <- DF_clean5_rows %>%
  ## Create columns with the first letter of each row
  mutate(
    Type.tmp      = str_sub(Type, 1, 1),
    Treatment.tmp = str_sub(Treatment, 1, 1)
  ) %>%
  ## Combine columns, and remove them
  unite(
    Plant,                             # new column name
    Type.tmp, Treatment.tmp, PlantNum, # input columns
    sep = ""      # characters to separate each input
  ) %>%
  ## Move columns to the front (left)
  relocate(Plant, Type, Treatment)
```

Long vs wide

- Our data frame is looking clean!
  But it still has one problem: it's in 'wide' format

```
DF_clean %>% select(where(is.numeric)) %>% names()
```

```
# [1] "X95"   "X175"  "X250"  "X350"  "X500"  "X675"  "X1000"
```

- All the numeric columns are values of a hidden variable, '`uptake`'
- The names of these columns are actually *values* of another hidden variable, '`conc`' (concentration)
- This is **not** 'tidy'
  - most plotting functions will not expect column names as values
  - analysis is more complicated: relationships are between the *column names* and *values*

Pivot

- We can make this data ‘tidy’ by *pivoting* to a longer structure
  - then convert the former column names to numeric values.

```
DF_tidy <- DF_clean %>%
  pivot_longer(
    cols = where(is.numeric),  # columns to pivot
    names_to = "conc",         # name of new column with old c
    values_to = "uptake"       # name of new column with old v
  ) %>%
  ## Clean former column names and convert to numeric
  mutate(
    conc = str_replace(conc, "X", "") %>% as.numeric()
  )
```

Check results

- Did we manage to re-construct the original data set?

```
all.equal(DF_tidy, CO2, check.attributes = FALSE)
```

```
# [1] "Component \"Plant\": target is character, current is ordered"
# [2] "Component \"Type\": target is character, current is factor"
# [3] "Component \"Treatment\": target is character, current is fact
```

- Not quite — our character columns are not '*factors*', as in the original.
  - *factors* are a special kind of vector for categorical data
  - See `?factor`, and the `forcats` package for more information

Final steps

- Convert all character columns to *factors*

```
DF_final <- DF_tidy %>%
  mutate( across(where(is.character), factor) )
```

- Did we manage to re-construct the original data set?

```
all.equal(DF_final, CO2, check.attributes = FALSE)
```

```
# [1] TRUE
```

Order of operations: clean, tidy

- Clean then tidy; or tidy then clean?
- It depends!
  - In this example, we cleaned columns before pivoting to combine them. This allowed us to correct *different* issues in each column, and convert them to a common type before combining.
  - If the same issue is present in multiple columns, it often makes sense to pivot first, then you have fewer columns to clean
  - In other cases, you may want to pivot *wider* to separate different variables, so that they can be cleaned differently.
- In many cases, you might switch between the two more than once.
  - e.g., clean $\rightarrow$ tidy $\rightarrow$ clean ...

Exercise

Section 14

Save Data Outside R

The readr package: writing data

readr		base R
write_csv()	← comma separated values	write.csv()
write_csv2()	← allows ';' as delimiter and ',' for decimals (depending on locale)	write.csv2() ',' for decimals, ';' as separator
write_tsv()	← tab separated values	
write_delim()	← (generic) files with an arbitrary delimiter	write.table()
write_excel_csv(), write_excel_cs2v()	← include a UTF-8 Byte order mark, which indicates to Excel the csv is UTF-8 encoded	

Save our work

- Save a data frame to a `.csv` file:
  - it will be encoded with `UTF-8` by default (on all platforms)

```
write_csv(DF_final, "data/data_clean.csv")
write_excel_csv(DF_final, "data/data_excel.csv")
```

Save our work

- Save a data frame to a .csv file:
  - it will be encoded with UTF-8 by default (on all platforms)

```
write_csv(DF_final, "data/data_clean.csv")
write_excel_csv(DF_final, "data/data_excel.csv")
```

- Read it back in to check

```
save_test <- read_csv("data/data_clean.csv")
head(save_test)
```

Section 15

Sharing Code

Style

"L'enfer, c'est les autres" ("Hell is other people")
— Jean-Paul Sartre ("Huis clos" / "No Exit")

"Hell is other people's code." — programming aphorism

- The syntax of the `R` language is strict about some things, but not others, like white space and indentation.
- As mentioned at the beginning, there is often more than one way to do things in `R`
  - different styles of *naming things*
  - different name formats: `camelCase`, `snake_case`, etc.
- Reading someone else's code that is written in a different style, or with inconsistent formatting, can be confusing.

Style Guides

- A "Style Guide" can be a useful tool to help you and your collaborators write code in a consistent style.
- It also simplifies writing code, by reducing the number of (style) decisions you have to make.
- A Style Guide is ***strongly*** *recommended* for teams collaborating on shared code.
  - Even if you are working alone, it can help you write cleaner code that's easy for your *future-self* to read and understand, and for others to help you when you get stuck.

Some popular R style guides you can use (or adapt):

- The tidyverse style guide
  - based on an earlier version of Google's style guide.
- Google's R Style Guide
  - based on the current tidyverse style guide, above.

Section 16

Review

Exercise

Quiz Review

Section 17

Backmatter

Other packages to look at

- `data.table`: a high-performance version of `data.frame` with few dependencies.

Other packages in the `tidyverse`:

- `lubridate` and `hms`: for date & time values.
- `purrr`: functional programming (FP) tools for working with functions and vectors.
  - Replace `for` loops with code that is more efficient and easier to read.

References

Cheatsheets:

- readr/readxl
- Data transformation with dplyr
- Data tidying with tidyr

On the web:

- Tidyverse documentation
- R for Data Science (2e)
- Data Science in a Box (#dsbox)
- An introduction to data cleaning with R

`R` Documentation:

- "R Data Import/Export" (`help.start()`, under "Manuals")