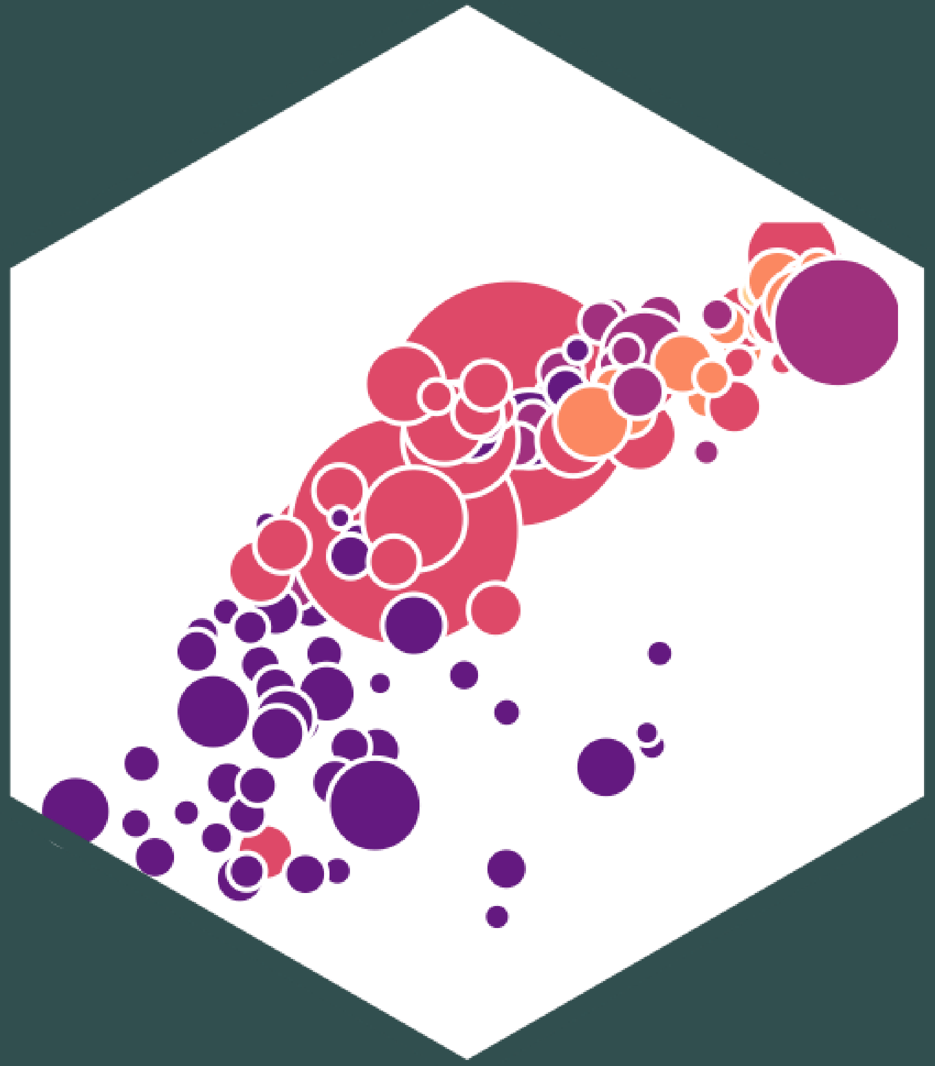
1.4 — Data Wrangling

ECON 480 • Econometrics • Fall 2022

Dr. Ryan Safner
Associate Professor of Economics

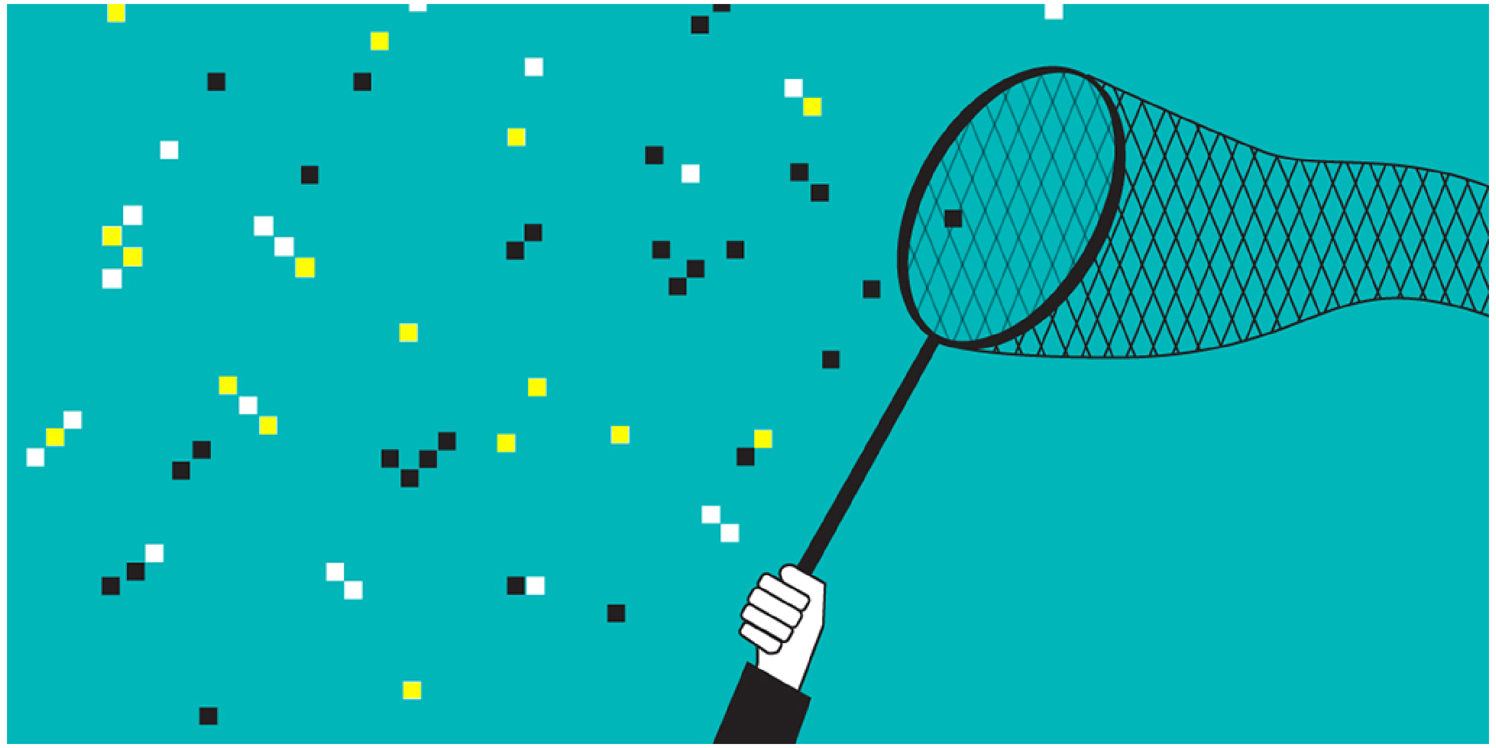
safner@hood.edu
ryansafner/metricsF22
metricsF22.classes.ryansafner.com

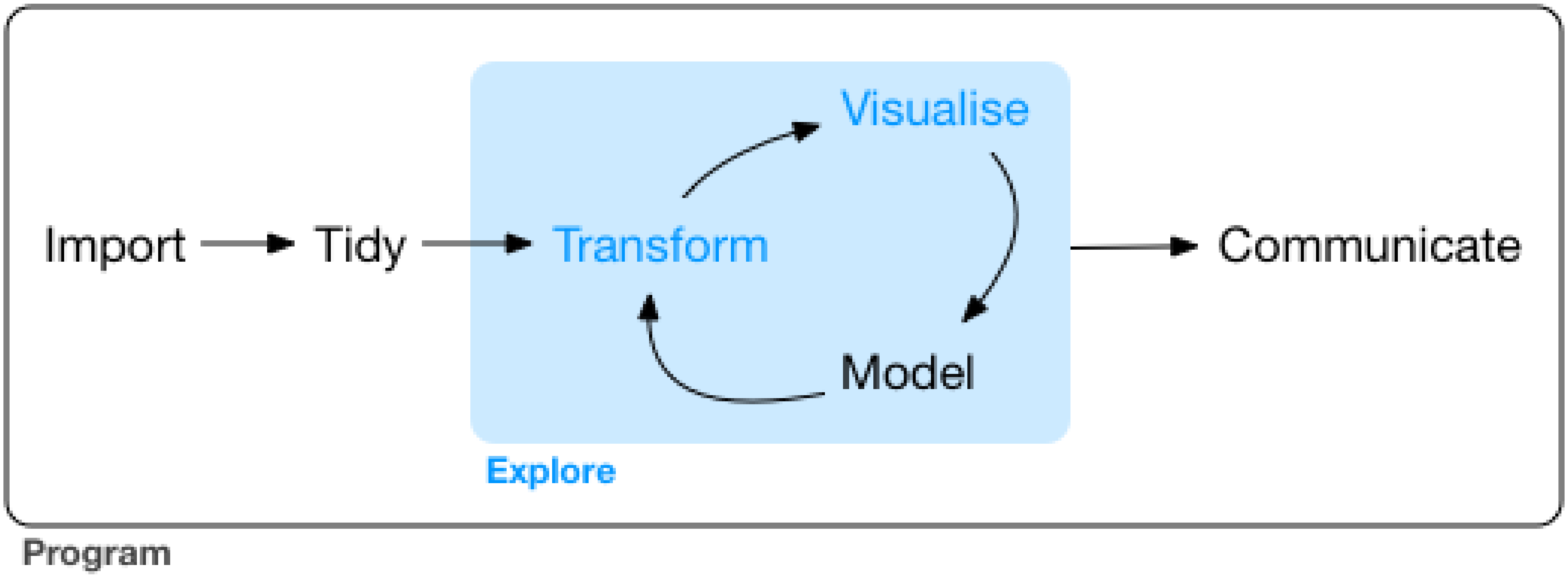


Contents

Data Wrangling

- Most data analysis is taming chaos into order
  - Data strewn from multiple sources 😨
  - Missing data (“NA”) 😡
  - Data not in a readable form 🤢

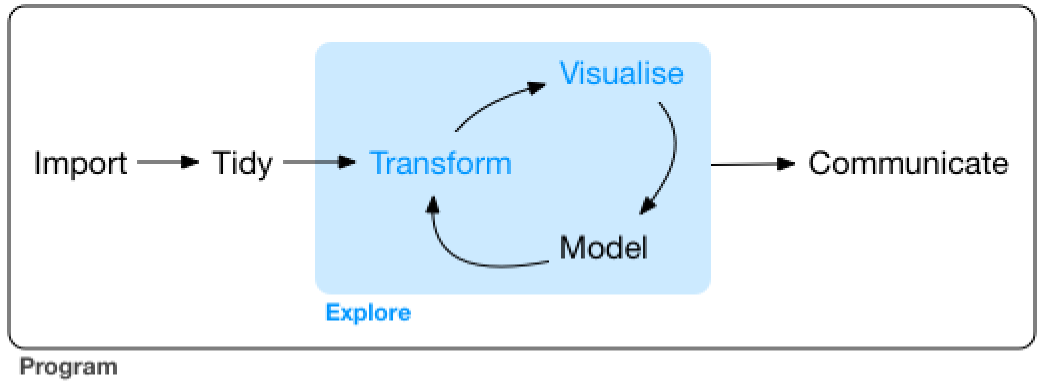



Import
Tidy
Transform
Visualise
Model
Explore
Communicate
Program

Workflow of a Data Scientist I

1. **Import** raw data from out there in the world
2. **Tidy** it into a form that you can use
3. **Explore** the data (do these 3 repetitively!)
    - **Transform**
    - **Visualize**
    - **Model**
4. **Communicate** results to target audience

Ideally, you'd want to be able to do all of this in **one program**



R for Data Science

Workflow of a Data Scientist II

The New York Times

For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights

Monica Rogati, Jawbone's vice president for data science, with Brian Wilt, a senior data scientist.
Peter DaSilva for The New York Times

By Steve Lohr

Aug. 17, 2014

Technology revolutions come in measured, sometimes foot-dragging steps. The lab science and marketing enthusiasm tend to

> "Yet far too much handcrafted work - what data scientists call"**data wrangling**," "**data munging**," and "**data janitor work**" - is still required. Data scientists, according to interviews and expert estimates, spend from **50 to 80 percent of their time** mired in this more mundane labor of collecting and preparing unruly digital data, before it can be explored for useful nuggets."

Source: New York Times

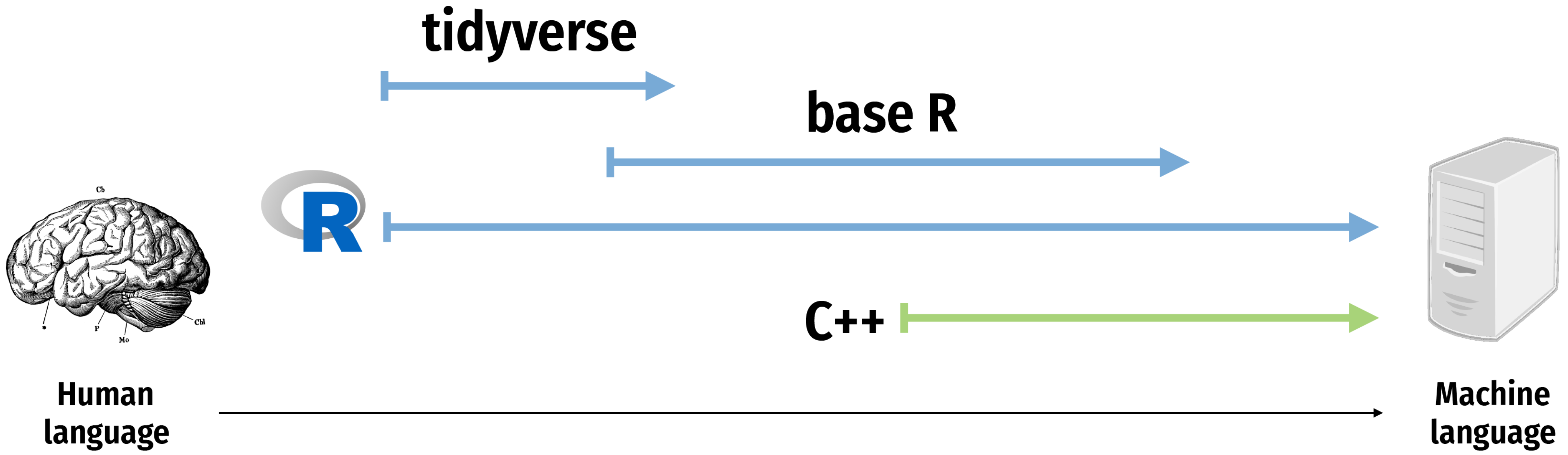
tidyverse

The `tidyverse` I

> "The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

- Allows you to do all of those things with one (set of) package(s)!
- Learn more at tidyverse.org

The `tidyverse` II



The `tidyverse` III

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

The `tidyverse` IV

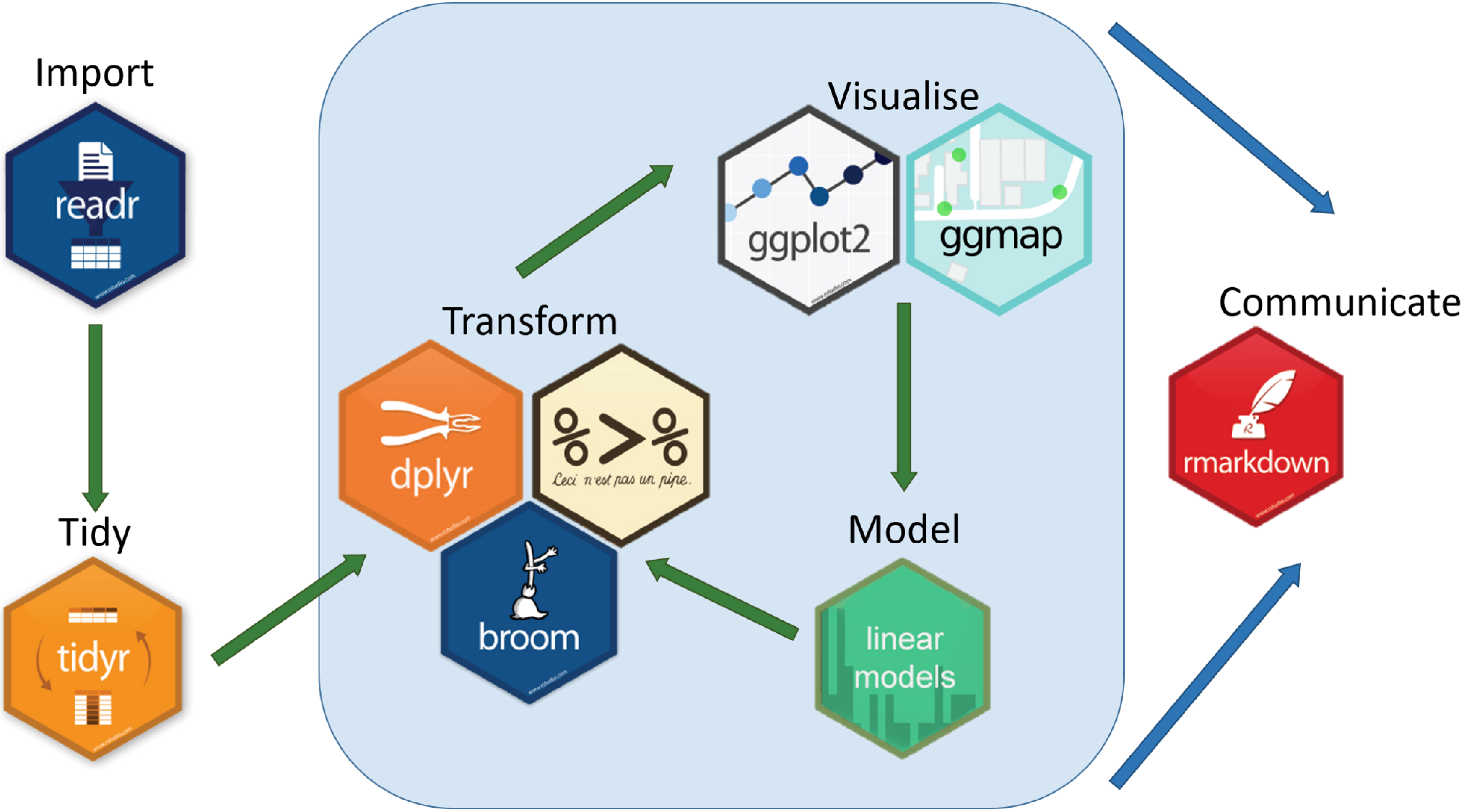
- `tidyverse` contains a lot of packages

```
1 tidyverse_packages()
```

```
 [1] "broom"         "cli"           "crayon"        "dbplyr"
 [5] "dplyr"         "dtplyr"        "forcats"       "googledrive"
 [9] "googlesheets4" "ggplot2"       "haven"         "hms"
[13] "httr"          "jsonlite"      "lubridate"     "magrittr"
[17] "modelr"        "pillar"        "purrr"         "readr"
[21] "readxl"        "reprex"        "rlang"         "rstudioapi"
[25] "rvest"         "stringr"       "tibble"        "tidyr"
[29] "xml2"          "tidyverse"
```

- Only the "core" packages are loaded automatically with `library(tidyverse)`:
  - `ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, forcats`

Your Workflow in the `tidyverse`:



Tibbles & Piping

Tibbles

- A `tibble` (or `tbl_df`) is a friendlier `data.frame`
- Fundamental grammar of tidyverse:
  1. start with a tibble
  2. run a function on it
  3. output a new tibble
- Loading `tidyverse` automatically converts all `data.frames` to `tibbles`

Tibbles: Example I

```r
# look at data
diamonds
```

```
# A tibble: 53,940 × 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# … with 53,930 more rows
```

Tibbles: Example II

```r
# another useful command
glimpse(diamonds)
```

```
Rows: 53,940
Columns: 10
$ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23, 0.…
$ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good, Ver…
$ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, J, J, I,…
$ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI1, VS1, …
$ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, 59.4, 64…
$ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62, 58…
$ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340, 34…
$ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, 4.00, 4.…
$ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, 4.05, 4.…
$ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, 2.39, 2.…
```

Tibbles: Making a Tibble

- Create a `tibble` from a `data.frame` with `as_tibble()`

```r
as_tibble(diamonds)
```

```
# A tibble: 53,940 × 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# … with 53,930 more rows
```

Tibbles: Making a Tibble (from Scratch)

- Create a `tibble` from scratch with `tibble()`, works like `data.frame()`

```r
example <- tibble(x = seq(2,6,2), # sequence from 2 to 6 by 2's
                  y = rnorm(3,0,1), # 3 random draws with mean 0, sd 1
                  colors = c("orange", "green", "blue")) # colors

example # look at it
```

```
# A tibble: 3 × 3
      x      y colors
  <dbl>  <dbl> <chr>
1     2 -0.902 orange
2     4  0.157 green
3     6 -0.396 blue
```

Tibbles: Making a Tibble (from Scratch)

- Create a `tibble` row by row with `tribble()`

```r
example_2 <- tribble(
  ~x, ~y, ~color, # each variable name must start with ~
  2, 1.5, "orange",
  4, 0.2, "green",
  6, 0.8, "blue") # last element has no comma

example_2 # look at it
```

```
# A tibble: 3 × 3
      x     y color
  <dbl> <dbl> <chr>
1     2   1.5 orange
2     4   0.2 green
3     6   0.8 blue
```

Piping Code

- The `magrittr` package allows use of the **"pipe" operator** (`%>%`)[1]
- `%>%` "pipes" the *output* of the *left* of the pipe *into* the *(1st)* *argument* of the *right*
- Running a function `f` on object `x` as `f(x)` becomes `x %>% f` in pipeable form
  - i.e. "take `x` and then run function `f` on it"

1. Keyboard shortcuts in R Studio: `CTRL+Shift+M` (Windows) or `Cmd+Shift+M` (Mac). There is also a *new* pipe built into Base R: `|>`

Piping Code

- With math functions, typically read from outside ← (inside):

Example

$$g(f(x))$$

take `x` and perform function `f()` on `x` and then perform function `g()` on that result

- With pipes, read operations from left → right:

```
x %>% f %>% g
```

- Can read `%>%` mentally as "and then"

Why Piping is Useful

Example

Get the average highway miles per gallon of Audi cars

Without pipes:

```r
summarize(group_by(filter(mpg, manufacturer == "audi"), model), hwy_avg = mean(hwy))
```

```
# A tibble: 3 × 2
  model      hwy_avg
  <chr>        <dbl>
1 a4            28.3
2 a4 quattro    25.8
3 a6 quattro    24
```

With pipes:

```r
mpg %>%
  filter(manufacturer == "audi") %>%
  group_by(model) %>%
  summarize(hwy_avg = mean(hwy))
```

```
# A tibble: 3 × 2
  model      hwy_avg
  <chr>        <dbl>
1 a4            28.3
```

Importing Data

Importing Data I

- Load common spreadsheet files (`.csv`, `.tsv`) with simple commands:
- `read_*(path/to/my_data.*)`
  - where `*` can be `.csv` or `.tsv`
- Can also *export* your data from R into a common spreadsheet file with:
- `write_*(my_df, path = path/to/file_name.*)`
  - where `my_df` is the name of your `tibble`, and `file_name` is the name of the file you want to save as
- Often this is enough, but much more customization possible
- Read more on the tidyverse website and the Readr Cheatsheet

Importing Data II

- For other data types from software programs like Excel, STATA, SAS, and SPSS:
- `readxl` has equivalent commands for Excel data types:
  - `read_*("path/to/my/data.*")`
  - `write_*(my_dataframe, path=path/to/file_name.*)`
  - where `*` can be `.xls` or `.xlsx`
- `haven` has equivalent commands for other data types:
  - `read_*("path/to/my_data.dta")` for STATA `.dta` files
  - `write_*(my_dataframe, path=path/to/file_name.*)`
  - where `*` can be `.dta` (STATA), `.sav` (SPSS), `.sas7bdat` (SAS)

Importing Data: Common Issues

- *"where the hell is my data file"??*
- Recall R looks for files to `read_*()` in the default working directory[1]
- You can tell R where this data is by making the `path` a part of the file's name when importing
  - Use `..` to "move up one folder"
  - Use `/` to "enter a folder"

1. Again, check what it is with `getwd()`, change it with `setwd()`

Aside: File Directories

- You can tell R where this data is by making the `path` a part of the file's name when importing
  - Use `..` to "move up one folder"
  - Use `/` to "enter a folder"
- Either use an **absolute path** on your computer:

```r
# Example

df <- read_csv("C:/Documents and Settings/Ryan Safner/Downloads/my_data.csv")
```

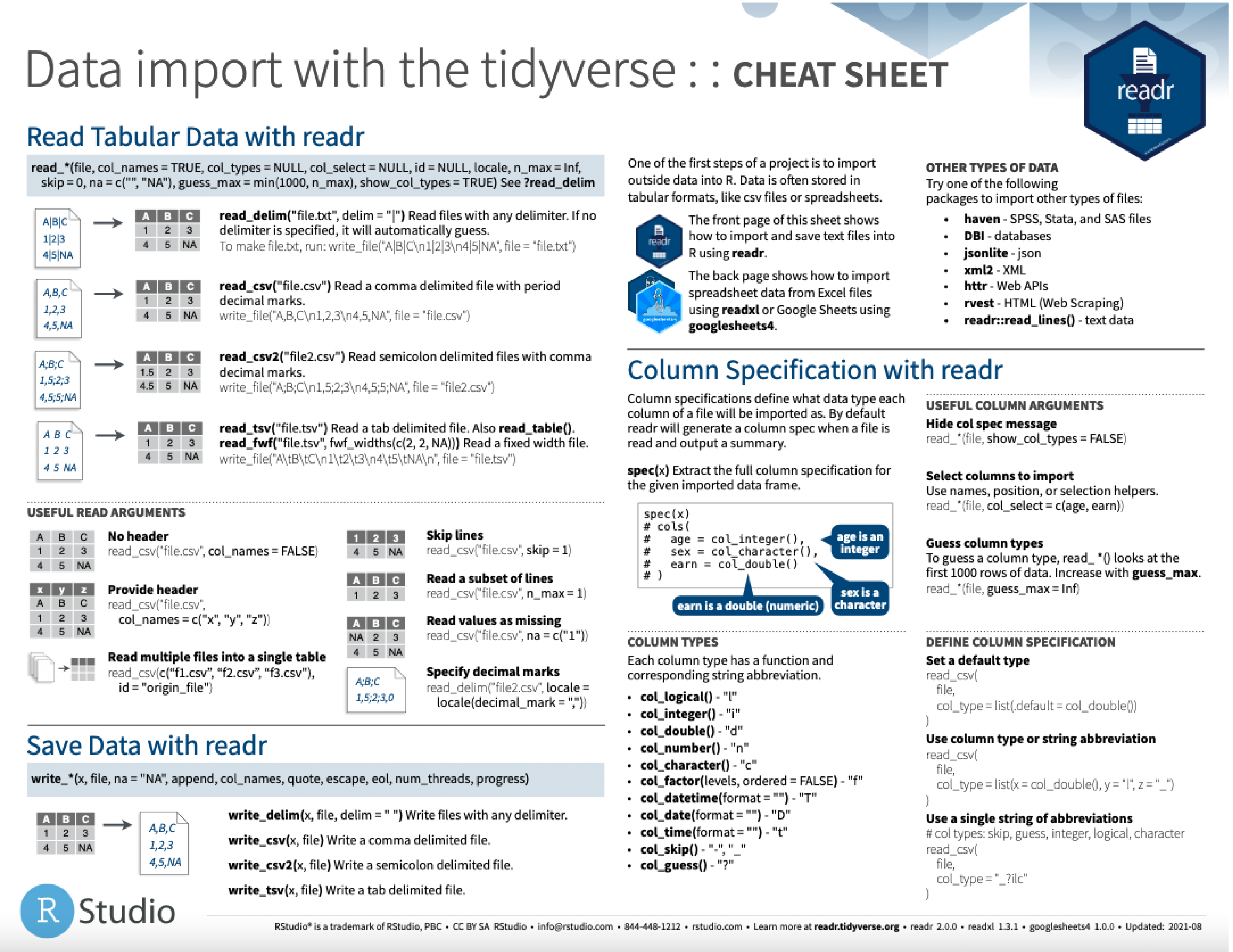
- Or use a **relative path** *from* R's working directory

```r
# Example
# If working directory is Documents, but data is in Downloads, like so:
#
# Ryan Safner/
# |
# |- Documents/
# |- Downloads/
# |- Photos/
# |- Videos/
df <- read_csv("../Downloads/my_data.csv")
```

Common Import Issues II

- **Suggestion** to make your data import easier: *Download and move files to R's working directory*
- Your computer and working directory are different from mine (and others)
- This is *not* a reproducible workflow!
- We'll finally fix this next class with `R Projects`
  - The working directory is set to the Project Folder by default
  - Same for everyone on any computer!

Data Import Cheat Sheet



Data import with the tidyverse : : CHEAT SHEET

readr

Read Tabular Data with readr

read_*(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf, skip = 0, na = c("", "NA"), guess_max = min(1000, n_max), show_col_types = TRUE) See ?read_delim

A	B	C
1	2	3
4	5	NA

read_delim("file.txt", delim = "|") Read files with any delimiter. If no delimiter is specified, it will automatically guess.
To make file.txt, run: write_file("A|B|C\n1|2|3\n4|5|NA", file = "file.txt")

read_csv("file.csv") Read a comma delimited file with period decimal marks.
write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")

read_csv2("file2.csv") Read semicolon delimited files with comma decimal marks.
write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")

read_tsv("file.tsv") Read a tab delimited file. Also **read_table()**.
read_fwf("file.tsv", fwf_widths(c(2, 2, NA))) Read a fixed width file.
write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA\n", file = "file.tsv")

USEFUL READ ARGUMENTS

No header
read_csv("file.csv", col_names = FALSE)

Provide header
read_csv("file.csv",
col_names = c("x", "y", "z"))

Read multiple files into a single table
read_csv(c("f1.csv", "f2.csv", "f3.csv"),
id = "origin_file")

Skip lines
read_csv("file.csv", skip = 1)

Read a subset of lines
read_csv("file.csv", n_max = 1)

Read values as missing
read_csv("file.csv", na = c("1"))

Specify decimal marks
read_delim("file2.csv", locale =
locale(decimal_mark = ","))

Save Data with readr

write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress)

write_delim(x, file, delim = " ") Write files with any delimiter.

write_csv(x, file) Write a comma delimited file.

write_csv2(x, file) Write a semicolon delimited file.

write_tsv(x, file) Write a tab delimited file.

One of the first steps of a project is to import outside data into R. Data is often stored in tabular formats, like csv files or spreadsheets.

The front page of this sheet shows how to import and save text files into R using **readr**.

The back page shows how to import spreadsheet data from Excel files using **readxl** or Google Sheets using **googlesheets4**.

OTHER TYPES OF DATA

Try one of the following packages to import other types of files:

- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

Column Specification with readr

Column specifications define what data type each column of a file will be imported as. By default readr will generate a column spec when a file is read and output a summary.

spec(x) Extract the full column specification for the given imported data frame.

```
spec(x)
# cols(
#   age = col_integer(),
#   sex = col_character(),
#   earn = col_double()
# )
```

age is an integer
sex is a character
earn is a double (numeric)

COLUMN TYPES

Each column type has a function and corresponding string abbreviation.

- **col_logical()** - "l"
- **col_integer()** - "i"
- **col_double()** - "d"
- **col_number()** - "n"
- **col_character()** - "c"
- **col_factor**(levels, ordered = FALSE) - "f"
- **col_datetime**(format = "") - "T"
- **col_date**(format = "") - "D"
- **col_time**(format = "") - "t"
- **col_skip()** - "-", "_"
- **col_guess()** - "?"

USEFUL COLUMN ARGUMENTS

Hide col spec message
read_*(file, show_col_types = FALSE)

Select columns to import
Use names, position, or selection helpers.
read_*(file, col_select = c(age, earn))

Guess column types
To guess a column type, read_ *() looks at the first 1000 rows of data. Increase with **guess_max**.
read_*(file, guess_max = Inf)

DEFINE COLUMN SPECIFICATION

Set a default type
```
read_csv(
  file,
  col_type = list(.default = col_double())
)
```

Use column type or string abbreviation
```
read_csv(
  file,
  col_type = list(x = col_double(), y = "l", z = "_")
)
```

Use a single string of abbreviations
```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

R Studio

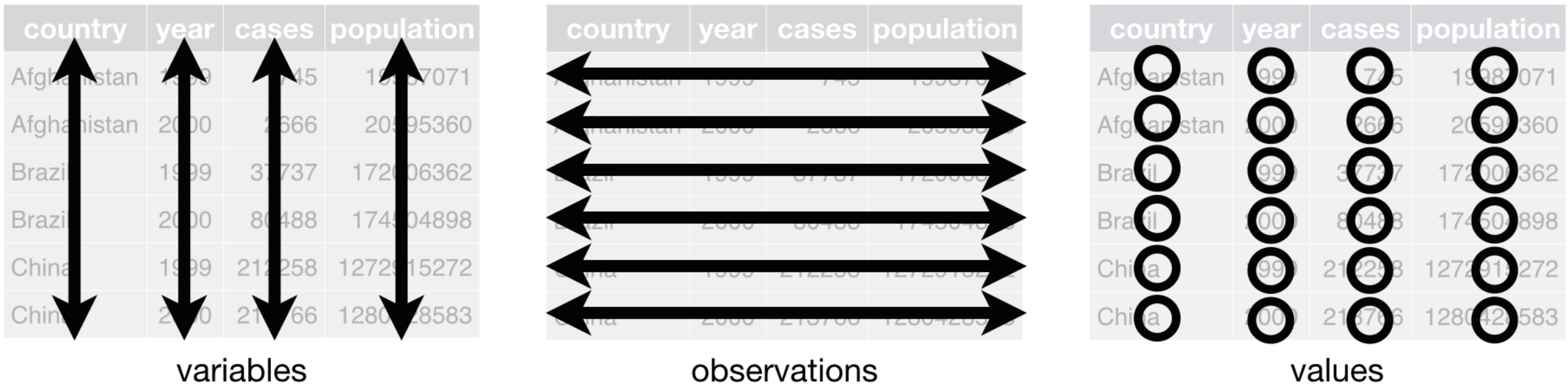
RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at readr.tidyverse.org • readr 2.0.0 • readxl 1.3.1 • googlesheets4 1.0.0 • Updated: 2021-08

R Studio: Data Import Cheat Sheet

Tidying (Pivoting/Reshaping) Data

Tidy Data

- **"tidy" data** are (an opinionated view of) data where
  1. Each **variable** is in a **column**
  2. Each **observation** is a **row**
  3. Each **observational unit** forms a **table** (or "every value is its own cell.")
- This is the namesake of the `tidyverse`: all associated packages and functions require tidy data
  - Spend less time fighting your tools and more time on analysis!

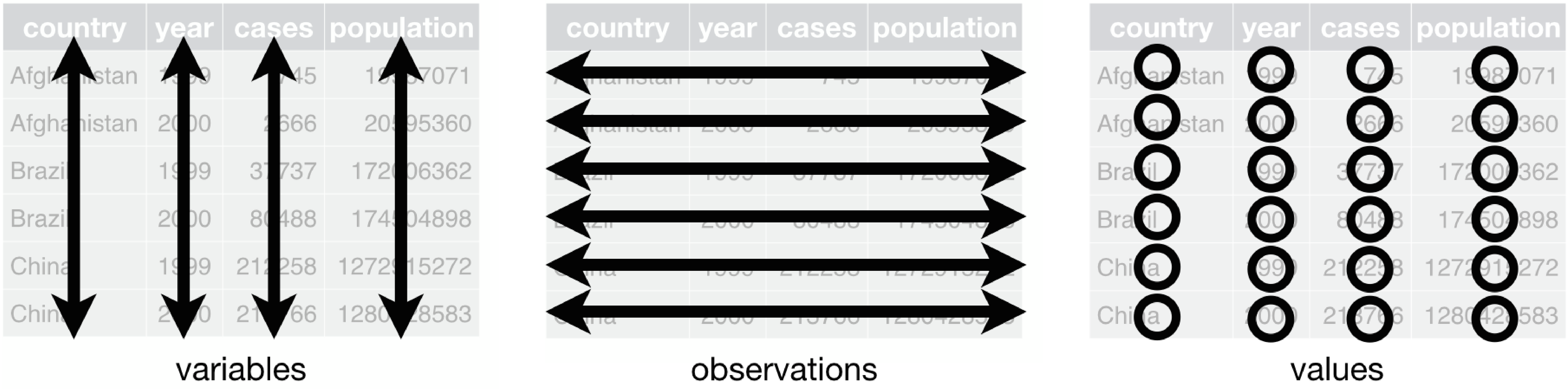


Tidy vs. Untidy Data

- “Tidy” data ≠ clean, perfect 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



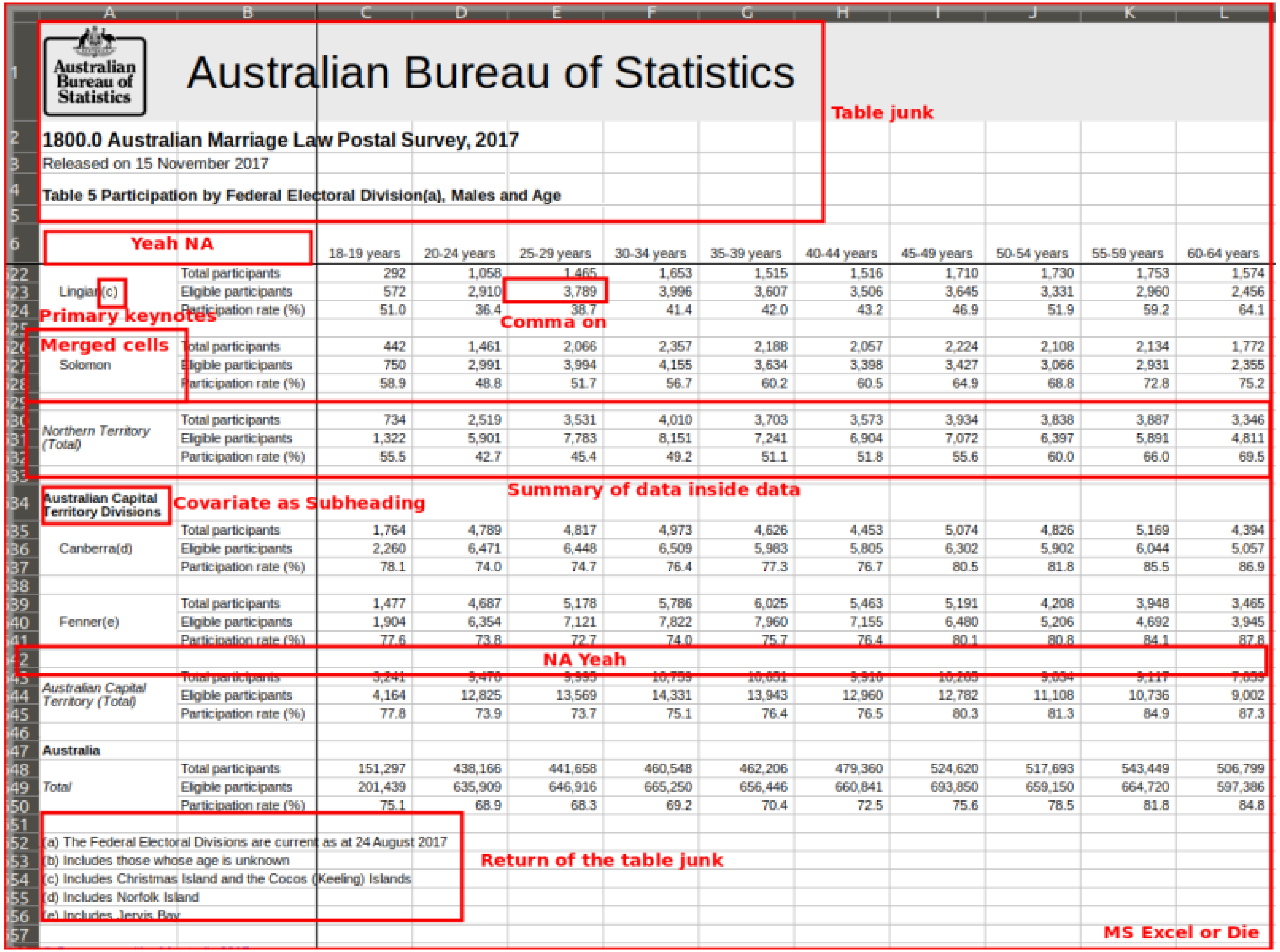
Examples of Untidy Data

	A	AA	AB	AC	AD	AE	AF	AG	AH
1	Estimated HIV Prevalence% - (Ages 15-49)	2004	2005	2006	2007	2008	2009	2010	2011
2	Abkhazia								
3	Afghanistan						0.06	0.06	0.06
4	Akrotiri and Dhekelia								
5	Albania								
6	Algeria	0.1	0.1	0.1	0.1	0.1			
7	American Samoa								
8	Andorra								
9	Angola	1.9	1.9	1.9	1.9	2	2.1	2.1	2.1
10	Anguilla								
11	Antigua and Barbuda								
12	Argentina	0.4	0.4	0.4	0.4	0.5	0.4	0.4	0.4
13	Armenia	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2
14	Aruba								
15	Australia	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2
16	Austria	0.2	0.2	0.2	0.3	0.3	0.3	0.4	0.4
17	Azerbaijan	0.06	0.06	0.06	0.1	0.1	0.1	0.1	0.1
18	Bahamas	3	3	3	3.1	3.1	2.9	2.8	2.8

Examples of Untidy Data

Subject	United States			
	Estimate	Margin of Error	Percent	Percent Margin of Error
EMPLOYMENT STATUS				
Population 16 years and over	255,797,692	+/-17,051	255,797,692	(X)
In labor force	162,184,325	+/-135,158	63.4%	+/-0.1
Civilian labor force	161,159,470	+/-127,501	63.0%	+/-0.1
Employed	150,599,165	+/-138,066	58.9%	+/-0.1
Unemployed	10,560,305	+/-27,385	4.1%	+/-0.1
Armed Forces	1,024,855	+/-10,363	0.4%	+/-0.1
Not in labor force	93,613,367	+/-126,007	36.6%	+/-0.1
Civilian labor force	161,159,470	+/-127,501	161,159,470	(X)
Unemployment Rate	(X)	(X)	6.6%	+/-0.1
Females 16 years and over	131,092,196	+/-11,187	131,092,196	(X)
In labor force	76,493,327	+/-75,824	58.4%	+/-0.1
Civilian labor force	76,350,498	+/-75,238	58.2%	+/-0.1
Employed	71,451,559	+/-79,007	54.5%	+/-0.1
Own children of the householder under 6 years	22,939,897	+/-14,240	22,939,897	(X)
All parents in family in labor force	14,957,537	+/-36,506	65.2%	+/-0.1
Own children of the householder 6 to 17 years	47,007,147	+/-19,644	47,007,147	(X)
All parents in family in labor force	33,238,793	+/-49,036	70.7%	+/-0.1

Examples of Untidy Data



Australian Bureau of Statistics

1800.0 Australian Marriage Law Postal Survey, 2017

Released on 15 November 2017

Table 5 Participation by Federal Electoral Division(a), Males and Age

		18-19 years	20-24 years	25-29 years	30-34 years	35-39 years	40-44 years	45-49 years	50-54 years	55-59 years	60-64 years
Lingiari(c)	Total participants	292	1,058	1,465	1,653	1,515	1,516	1,710	1,730	1,753	1,574
	Eligible participants	572	2,910	3,789	3,996	3,607	3,506	3,645	3,331	2,960	2,456
	Participation rate (%)	51.0	36.4	38.7	41.4	42.0	43.2	46.9	51.9	59.2	64.1
Solomon	Total participants	442	1,461	2,066	2,357	2,188	2,057	2,224	2,108	2,134	1,772
	Eligible participants	750	2,991	3,994	4,155	3,634	3,398	3,427	3,066	2,931	2,355
	Participation rate (%)	58.9	48.8	51.7	56.7	60.2	60.5	64.9	68.8	72.8	75.2
Northern Territory (Total)	Total participants	734	2,519	3,531	4,010	3,703	3,573	3,934	3,838	3,887	3,346
	Eligible participants	1,322	5,901	7,783	8,151	7,241	6,904	7,072	6,397	5,891	4,811
	Participation rate (%)	55.5	42.7	45.4	49.2	51.1	51.8	55.6	60.0	66.0	69.5
Australian Capital Territory Divisions											
Canberra(d)	Total participants	1,764	4,789	4,817	4,973	4,626	4,453	5,074	4,826	5,169	4,394
	Eligible participants	2,260	6,471	6,448	6,509	5,983	5,805	6,302	5,902	6,044	5,057
	Participation rate (%)	78.1	74.0	74.7	76.4	77.3	76.7	80.5	81.8	85.5	86.9
Fenner(e)	Total participants	1,477	4,687	5,178	5,786	6,025	5,463	5,191	4,208	3,948	3,465
	Eligible participants	1,904	6,354	7,121	7,822	7,960	7,155	6,480	5,206	4,692	3,945
	Participation rate (%)	77.6	73.8	72.7	74.0	75.7	76.4	80.1	80.8	84.1	87.8
Australian Capital Territory (Total)	Total participants	3,241	9,470	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	9,117	[illegible]
	Eligible participants	4,164	12,825	13,569	14,331	13,943	12,960	12,782	11,108	10,736	9,002
	Participation rate (%)	77.8	73.9	73.7	75.1	76.4	76.5	80.3	81.3	84.9	87.3
Australia											
Total	Total participants	151,297	438,166	441,658	460,548	462,206	479,360	524,620	517,693	543,449	506,799
	Eligible participants	201,439	635,909	646,916	665,250	656,446	660,841	693,850	659,150	664,720	597,386
	Participation rate (%)	75.1	68.9	68.3	69.2	70.4	72.5	75.6	78.5	81.8	84.8

(a) The Federal Electoral Divisions are current as at 24 August 2017
(b) Includes those whose age is unknown
(c) Includes Christmas Island and the Cocos (Keeling) Islands
(d) Includes Norfolk Island
(e) Includes Jervis Bay

Reshaping/Pivoting Data

- `tidyr` package helps reshape data into more usable format
- Most common use: reshaping data between “long” and “wide”

wide

id	x	y	z
1	a	c	e
2	b	d	f

long

id	key	val
1	x	a
2	x	b
1	y	c
2	y	d
1	z	e
2	z	f

Reshaping

`wide`

id	x	y	z
1	a	c	e
2	b	d	f

Source: Garrick Aden-Buie's tidyexplain

Reshaping from Wide to Long: `pivot_longer()` I

```
1  ex_wide
```

```
# A tibble: 3 × 3
  Country       `2000` `2010`
  <chr>          <dbl>  <dbl>
1 United States    140    180
2 Canada           102     98
3 China            111    123
```

- **Common source of "un-tidy" data**: **Column headers are values, not variable names!** 😨
  - Column names are *values* of a `year` variable! (e.g. `2000`, `2010`)
  - Each row actually represents *two* observations (one in 2000 and one in 2010)!

Reshaping from Wide to Long: `pivot_longer()` II

```
1  ex_wide
```

```
# A tibble: 3 × 3
  Country       `2000` `2010`
  <chr>          <dbl>  <dbl>
1 United States    140    180
2 Canada           102     98
3 China            111    123
```

- We need to `pivot_longer()` these columns into a new pair of variables to make a longer dataframe
  - set of columns represent *values* of one variable (`year`), not variables themselves! (`2000` and `2010`)
  - `names_to`: name of variable to create whose values form the column names (the “names” `2000` and `2010` are values of `year`)
  - `values_to`: name of the variable to create whose values are spread over the cells (we’ll call it number of `cases` for each country in each year)

Reshaping from Wide to Long: `pivot_longer()` III

- `pivot_longer()` a wide data frame into a long data frame

```r
ex_wide
```

```
# A tibble: 3 × 3
  Country       `2000` `2010`
  <chr>          <dbl>  <dbl>
1 United States    140    180
2 Canada           102     98
3 China            111    123
```

```r
ex_wide %>%
  pivot_longer(c("2000","2010"), # select columns
               names_to = "year", # variable for column names
               values_to = "cases") # values
```

```
# A tibble: 6 × 3
  Country       year  cases
  <chr>         <chr> <dbl>
1 United States 2000    140
2 United States 2010    180
3 Canada        2000    102
4 Canada        2010     98
5 China         2000    111
6 China         2010    123
```

Reshaping from Long to Wide: `pivot_wider()` I

```r
ex_long
```

```
# A tibble: 12 × 4
   country        year type       count
   <chr>         <dbl> <chr>      <dbl>
 1 United States  2000 cases        140
 2 United States  2000 population   300
 3 United States  2010 cases        180
 4 United States  2010 population   310
 5 Canada         2000 cases        102
 6 Canada         2000 population   110
 7 Canada         2010 cases         98
 8 Canada         2010 population   121
 9 China          2000 cases        111
10 China          2000 population  1201
11 China          2010 cases        123
```

- **Another common source of "un-tidy" data: observations are scattered across multiple rows!** 😨
  - Each country-year has two rows per observation, one for `Cases` and one for `Population` (categorized by `type` of variable)

Reshaping from Wide to Long: `pivot_wider()` II

```r
ex_long
```

```
# A tibble: 12 × 4
   country        year type       count
   <chr>         <dbl> <chr>      <dbl>
 1 United States  2000 cases        140
 2 United States  2000 population   300
 3 United States  2010 cases        180
 4 United States  2010 population   310
 5 Canada         2000 cases        102
 6 Canada         2000 population   110
 7 Canada         2010 cases         98
 8 Canada         2010 population   121
 9 China          2000 cases        111
10 China          2000 population  1201
11 China          2010 cases        123
```

- We need to `pivot_wider()` these columns into a new pair of variables
  - `names_from`: column that contains variable names (here, the `type`)
  - `values_from`: column that contains values from multiple variables (here, the `count`)

Reshaping from Wide to Long: `pivot_wider()` III

- `pivot_wider()` a long data frame into a wide data frame

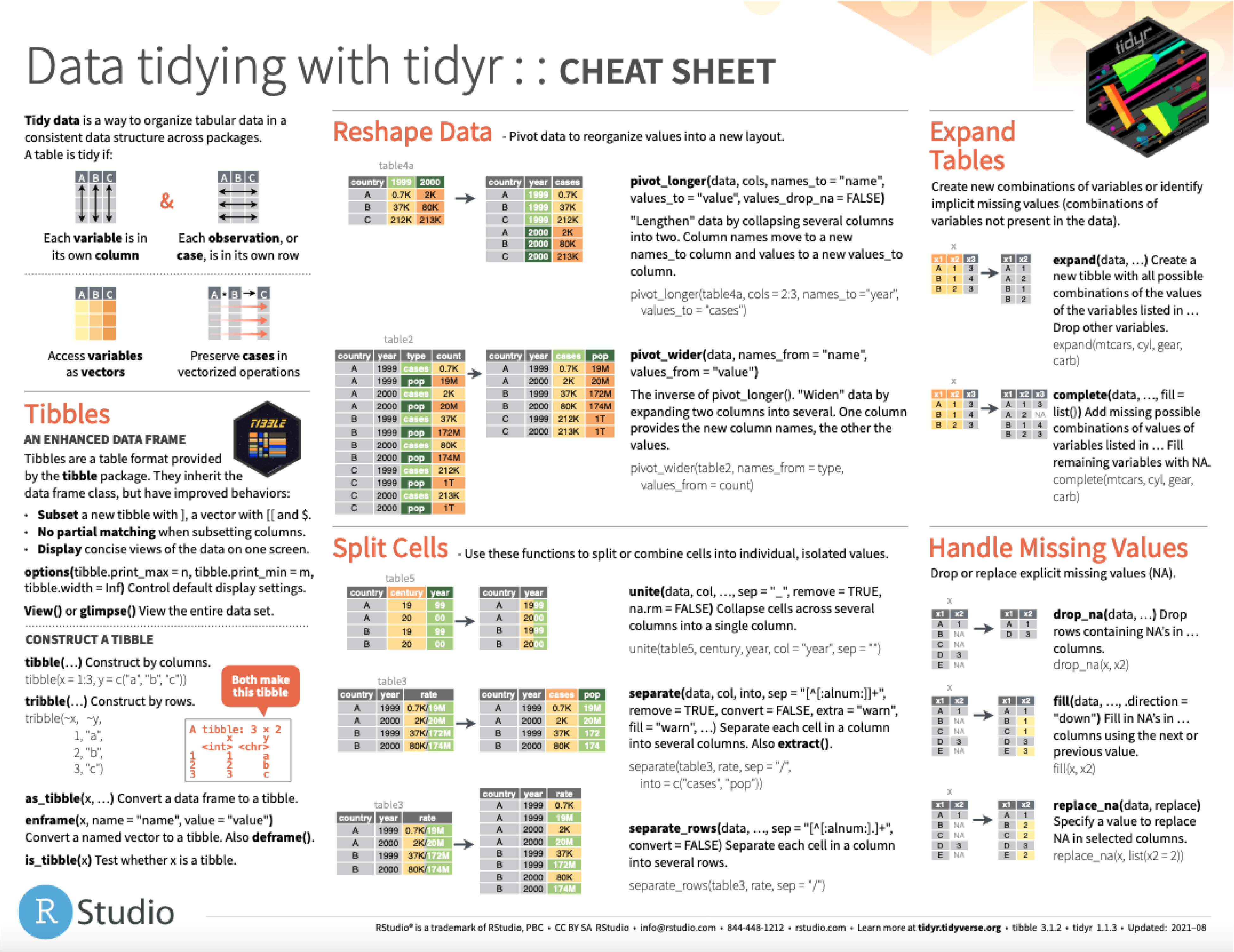
```r
ex_long
```

```
# A tibble: 12 × 4
   country        year type       count
   <chr>         <dbl> <chr>      <dbl>
 1 United States  2000 cases        140
 2 United States  2000 population   300
 3 United States  2010 cases        180
 4 United States  2010 population   310
 5 Canada         2000 cases        102
 6 Canada         2000 population   110
 7 Canada         2010 cases         98
 8 Canada         2010 population   121
 9 China          2000 cases        111
10 China          2000 population  1201
11 China          2010 cases        123
```

```r
ex_long %>%
  pivot_wider(names_from = "type", # column with names of var
              values_from = "count") # column with values of
```

```
# A tibble: 6 × 4
  country        year cases population
  <chr>         <dbl> <dbl>      <dbl>
1 United States  2000   140        300
2 United States  2010   180        310
3 Canada         2000   102        110
4 Canada         2010    98        121
5 China          2000   111       1201
6 China          2010   123       1241
```

Data Tidying Cheat Sheet


Data tidying with tidyr : : CHEAT SHEET

Tidy data is a way to organize tabular data in a consistent data structure across packages.
A table is tidy if:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own row

Access **variables** as **vectors**

Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

Tibbles are a table format provided by the **tibble** package. They inherit the data frame class, but have improved behaviors:

- **Subset** a new tibble with], a vector with [[and $.
- **No partial matching** when subsetting columns.
- **Display** concise views of the data on one screen.

options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf) Control default display settings.

View() or **glimpse**() View the entire data set.

CONSTRUCT A TIBBLE

tibble(…) Construct by columns.
tibble(x = 1:3, y = c("a", "b", "c"))

tribble(…) Construct by rows.
tribble(~x, ~y,
1, "a",
2, "b",
3, "c")

Both make this tibble

```
A tibble: 3 × 2
    x     y
<int> <chr>
1   1     a
2   2     b
3   3     c
```

as_tibble(x, …) Convert a data frame to a tibble.

enframe(x, name = "name", value = "value") Convert a named vector to a tibble. Also **deframe**().

is_tibble(x) Test whether x is a tibble.

Reshape Data

- Pivot data to reorganize values into a new layout.

pivot_longer(data, cols, names_to = "name", values_to = "value", values_drop_na = FALSE)

"Lengthen" data by collapsing several columns into two. Column names move to a new names_to column and values to a new values_to column.

pivot_longer(table4a, cols = 2:3, names_to ="year", values_to = "cases")

pivot_wider(data, names_from = "name", values_from = "value")

The inverse of pivot_longer(). "Widen" data by expanding two columns into several. One column provides the new column names, the other the values.

pivot_wider(table2, names_from = type, values_from = count)

Split Cells

- Use these functions to split or combine cells into individual, isolated values.

unite(data, col, …, sep = "_", remove = TRUE, na.rm = FALSE) Collapse cells across several columns into a single column.

unite(table5, century, year, col = "year", sep = "")

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", …) Separate each cell in a column into several columns. Also **extract**().

separate(table3, rate, sep = "/", into = c("cases", "pop"))

separate_rows(data, …, sep = "[^[:alnum:].]+", convert = FALSE) Separate each cell in a column into several rows.

separate_rows(table3, rate, sep = "/")

Expand Tables

Create new combinations of variables or identify implicit missing values (combinations of variables not present in the data).

expand(data, …) Create a new tibble with all possible combinations of the values of the variables listed in … Drop other variables.
expand(mtcars, cyl, gear, carb)

complete(data, …, fill = list()) Add missing possible combinations of values of variables listed in … Fill remaining variables with NA.
complete(mtcars, cyl, gear, carb)

Handle Missing Values

Drop or replace explicit missing values (NA).

drop_na(data, …) Drop rows containing NA's in … columns.
drop_na(x, x2)

fill(data, …, .direction = "down") Fill in NA's in … columns using the next or previous value.
fill(x, x2)

replace_na(data, replace) Specify a value to replace NA in selected columns.
replace_na(x, list(x2 = 2))

RStudio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **tidyr.tidyverse.org** • tibble 3.1.2 • tidyr 1.1.3 • Updated: 2021-08

R Studio: Data Tidying Cheat Sheet

Joining Datasets

Wrangling Data

dplyr

- `dplyr` uses more efficient & intuitive commands to manipulate tibbles
- `Base R` grammar passively runs functions on nouns: `function(object)`
- `dplyr` grammar actively uses verbs: `verb(df, conditions)`[1]

1. With the pipe, even simpler: `df %>% verb(conditions)`

`dplyr` II

- Great features:

  1. Allows use of `%>%` pipe operator
  2. Input and output is always a `tibble`
  3. Shows the output from a manipulation, but does not save/overwrite as an object unless explicitly assigned to an object
  4. Several packages provide backends to SQL (`dbplyr`), Apache Spark (`sparklyr`)

`dplyr` Verbs

- Common `dplyr` verbs

Verb	**Does**
`filter()`	Keep only selected *observations*
`select()`	Keep only selected *variables*
`arrange()`	Reorder rows (e.g. in numerical order)
`mutate()`	Create new variables
`summarize()`	Collapse data into summary statistics
`group_by()`	Perform any of the above functions by groups/categories

`arrange()`: Reorder observations

arrange()

- `arrange` reorders **observations** (rows) in a logical order
  - e.g. alphabetical, numeric, small to large

```r
# order by smallest to largest pop
gapminder %>%
  arrange(pop)
```

```
# A tibble: 1,704 × 6
   country               continent  year lifeExp   pop gdpPercap
   <fct>                 <fct>     <int>   <dbl> <int>     <dbl>
 1 Sao Tome and Principe Africa     1952    46.5 60011      880.
 2 Sao Tome and Principe Africa     1957    48.9 61325      861.
 3 Djibouti              Africa     1952    34.8 63149     2670.
 4 Sao Tome and Principe Africa     1962    51.9 65345     1072.
 5 Sao Tome and Principe Africa     1967    54.4 70787     1385.
 6 Djibouti              Africa     1957    37.3 71851     2865.
 7 Sao Tome and Principe Africa     1972    56.5 76595     1533.
 8 Sao Tome and Principe Africa     1977    58.6 86796     1738.
 9 Djibouti              Africa     1962    39.7 89898     3021.
10 Sao Tome and Principe Africa     1982    60.4 98593     1890.
# … with 1,694 more rows
```

`arrange()`: Ties

- Break ties in the value of one variable with the values of additional variables

```r
# order by year, with the smallest to largest pop in each year
gapminder %>%
  arrange(year, pop)
```

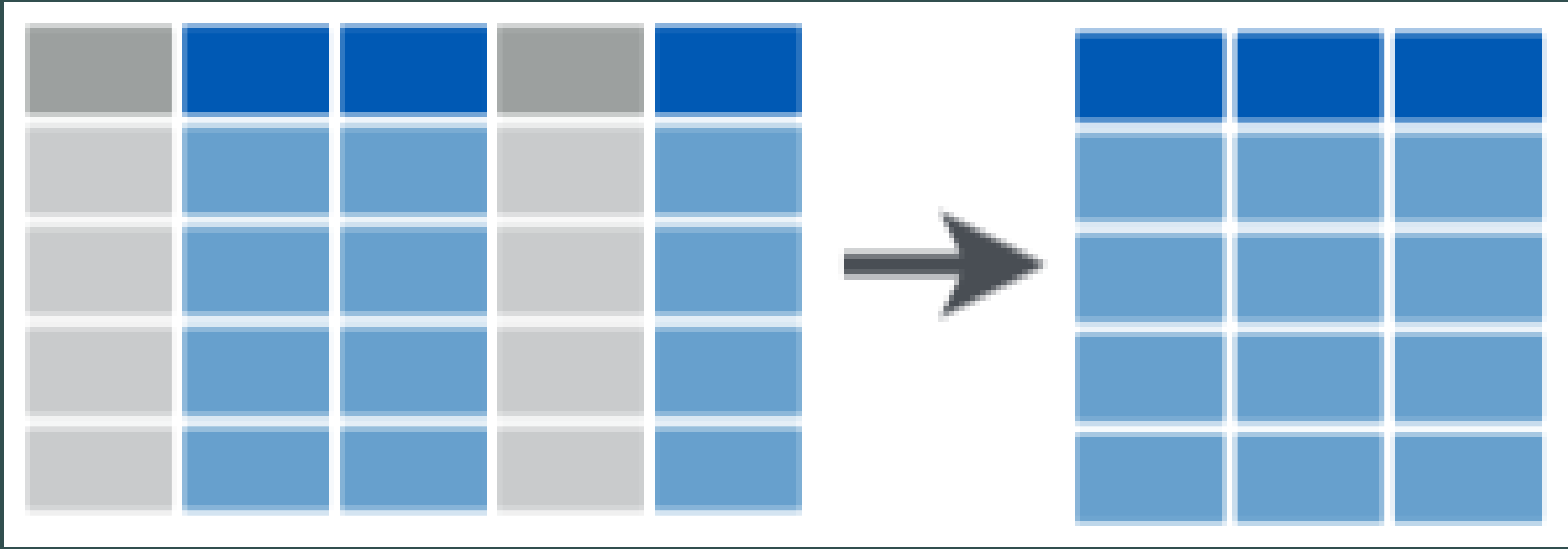
```
# A tibble: 1,704 × 6
   country               continent  year lifeExp    pop gdpPercap
   <fct>                 <fct>     <int>   <dbl>  <int>     <dbl>
 1 Sao Tome and Principe Africa     1952    46.5  60011      880.
 2 Djibouti              Africa     1952    34.8  63149     2670.
 3 Bahrain               Asia       1952    50.9 120447     9867.
 4 Iceland               Europe     1952    72.5 147962     7268.
 5 Comoros               Africa     1952    40.7 153936     1103.
 6 Kuwait                Asia       1952    55.6 160000   108382.
 7 Equatorial Guinea     Africa     1952    34.5 216964      376.
 8 Reunion               Africa     1952    52.7 257700     2719.
 9 Gambia                Africa     1952    30   284320      485.
10 Swaziland             Africa     1952    41.4 290243     1148.
# … with 1,694 more rows
```

`arrange()`: Descending Order

- Wrap `desc()` around a variable re-order in the opposite direction

```
# order by largest to smallest pop
gapminder %>%
  arrange(desc(pop))
```

`select()` Variables



select()

- `select` keeps only selected **variables** (columns)
  - Don't need quotes around column names

```r
# keep only country, year, and population variables
gapminder %>%
  select(country, year, pop)
```

```
# A tibble: 1,704 × 3
   country      year      pop
   <fct>       <int>    <int>
 1 Afghanistan  1952  8425333
 2 Afghanistan  1957  9240934
 3 Afghanistan  1962 10267083
 4 Afghanistan  1967 11537966
 5 Afghanistan  1972 13079460
 6 Afghanistan  1977 14880372
 7 Afghanistan  1982 12881816
 8 Afghanistan  1987 13867957
 9 Afghanistan  1992 16317921
10 Afghanistan  1997 22227415
# … with 1,694 more rows
```

`select()` *except*

- `select` "all except" by negating a variable with `-`

```
# keep all variables *except* gdpPercap
gapminder %>%
  select(-gdpPercap)
```

```
# A tibble: 1,704 × 5
   country     continent  year lifeExp      pop
   <fct>       <fct>     <int>   <dbl>    <int>
 1 Afghanistan Asia       1952    28.8  8425333
 2 Afghanistan Asia       1957    30.3  9240934
 3 Afghanistan Asia       1962    32.0 10267083
 4 Afghanistan Asia       1967    34.0 11537966
 5 Afghanistan Asia       1972    36.1 13079460
 6 Afghanistan Asia       1977    38.4 14880372
 7 Afghanistan Asia       1982    39.9 12881816
 8 Afghanistan Asia       1987    40.8 13867957
 9 Afghanistan Asia       1992    41.7 16317921
10 Afghanistan Asia       1997    41.8 22227415
# … with 1,694 more rows
```

`select()`: Reordering columns

- `select` reorders the columns in the order you provide
  - sometimes useful to keep all variables, and drag one or a few to the front, add `everything()` at the end

```r
# move pop to first column
gapminder %>%
  select(pop, everything())
```

```
# A tibble: 1,704 × 6
        pop country     continent  year lifeExp gdpPercap
      <int> <fct>       <fct>     <int>   <dbl>     <dbl>
 1  8425333 Afghanistan Asia       1952    28.8      779.
 2  9240934 Afghanistan Asia       1957    30.3      821.
 3 10267083 Afghanistan Asia       1962    32.0      853.
 4 11537966 Afghanistan Asia       1967    34.0      836.
 5 13079460 Afghanistan Asia       1972    36.1      740.
 6 14880372 Afghanistan Asia       1977    38.4      786.
 7 12881816 Afghanistan Asia       1982    39.9      978.
 8 13867957 Afghanistan Asia       1987    40.8      852.
 9 16317921 Afghanistan Asia       1992    41.7      649.
10 22227415 Afghanistan Asia       1997    41.8      635.
# … with 1,694 more rows
```

`select()` Helper Functions

- `select` has a lot of helper functions, useful for when you have hundreds of variables
  - see `?select()` for a list

```r
# keep all variables starting with "co"
gapminder %>%
  select(starts_with("co"))
```

```
# A tibble: 1,704 × 2
   country     continent
   <fct>       <fct>    
 1 Afghanistan Asia     
 2 Afghanistan Asia     
 3 Afghanistan Asia     
 4 Afghanistan Asia     
 5 Afghanistan Asia     
 6 Afghanistan Asia     
 7 Afghanistan Asia     
 8 Afghanistan Asia     
 9 Afghanistan Asia     
10 Afghanistan Asia     
# … with 1,694 more rows
```

`select()` Helper Functions

- `select` has a lot of helper functions, useful for when you have hundreds of variables
  - see `?select()` for a list

```
# keep country and all variables containing "per"
gapminder %>%
  select(country, contains("per"))
```

```
# A tibble: 1,704 × 2
   country     gdpPercap
   <fct>           <dbl>
 1 Afghanistan      779.
 2 Afghanistan      821.
 3 Afghanistan      853.
 4 Afghanistan      836.
 5 Afghanistan      740.
 6 Afghanistan      786.
 7 Afghanistan      978.
 8 Afghanistan      852.
 9 Afghanistan      649.
10 Afghanistan      635.
# … with 1,694 more rows
```

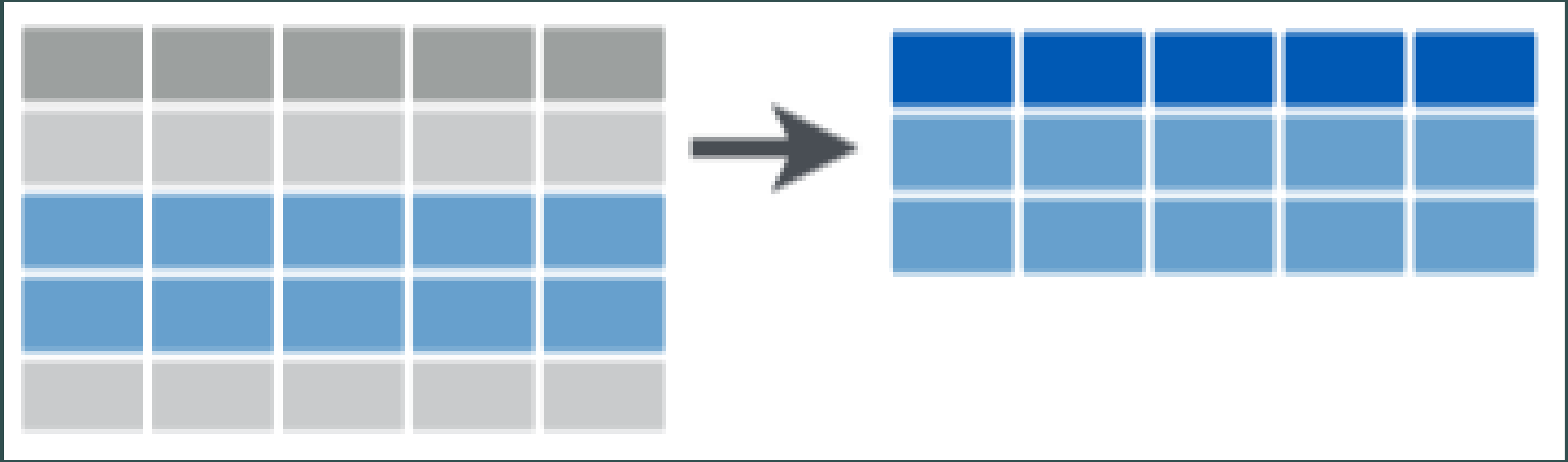
`rename()` Variables

- `rename` changes the name of a variable (column)
  - Format: `new_name = old_name`

```r
# rename gdpPercap to GDP and lifeExp to population
gapminder %>%
  rename(GDP = gdpPercap,
         LE = lifeExp)
```

```
# A tibble: 1,704 × 6
   country     continent  year    LE      pop   GDP
   <fct>       <fct>     <int> <dbl>    <int> <dbl>
 1 Afghanistan Asia       1952  28.8  8425333  779.
 2 Afghanistan Asia       1957  30.3  9240934  821.
 3 Afghanistan Asia       1962  32.0 10267083  853.
 4 Afghanistan Asia       1967  34.0 11537966  836.
 5 Afghanistan Asia       1972  36.1 13079460  740.
 6 Afghanistan Asia       1977  38.4 14880372  786.
 7 Afghanistan Asia       1982  39.9 12881816  978.
 8 Afghanistan Asia       1987  40.8 13867957  852.
 9 Afghanistan Asia       1992  41.7 16317921  649.
10 Afghanistan Asia       1997  41.8 22227415  635.
# … with 1,694 more rows
```

`filter()` Select Rows by Condition



filter()

- `filter` keeps only selected **observations** (rows)

```r
# look only at African observations
gapminder %>%
  filter(continent == "Africa")
```

```
# A tibble: 624 × 6
   country continent  year lifeExp      pop gdpPercap
   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
 1 Algeria Africa     1952    43.1  9279525     2449.
 2 Algeria Africa     1957    45.7 10270856     3014.
 3 Algeria Africa     1962    48.3 11000948     2551.
 4 Algeria Africa     1967    51.4 12760499     3247.
 5 Algeria Africa     1972    54.5 14760787     4183.
 6 Algeria Africa     1977    58.0 17152804     4910.
 7 Algeria Africa     1982    61.4 20033753     5745.
 8 Algeria Africa     1987    65.8 23254956     5681.
 9 Algeria Africa     1992    67.7 26298373     5023.
10 Algeria Africa     1997    69.2 29072015     4797.
# … with 614 more rows
```

Conditionals in R

- In many data wrangling contexts, you will want to select data **conditionally**
  - To a computer: observations for which a set of logical conditions are `TRUE`[1]

`>`	greater than	`<`	less than
`>=`	greater than or equal to	`<=`	less than or equal to
`==`[2]	is equal to	`!=`	is not equal to
`&`	and	`|`	or
`%in%`	is member of	`%notin%`	is not a member of

1. See `?Comparison` and `?Base::Logic`.
2. Recall one `=` *assigns* values to an object, two `==` *tests* an object for a condition!

`filter()` with Conditionals I

- Can chain multiple conditions with a `,`

```r
# look only at African observations in 1997
gapminder %>%
  filter(continent == "Africa",
         year == 1997)
```

```
# A tibble: 52 × 6
   country                  continent  year lifeExp      pop gdpPercap
   <fct>                    <fct>     <int>   <dbl>    <int>     <dbl>
 1 Algeria                  Africa     1997    69.2 29072015     4797.
 2 Angola                   Africa     1997    41.0  9875024     2277.
 3 Benin                    Africa     1997    54.8  6066080     1233.
 4 Botswana                 Africa     1997    52.6  1536536     8647.
 5 Burkina Faso             Africa     1997    50.3 10352843      946.
 6 Burundi                  Africa     1997    45.3  6121610      463.
 7 Cameroon                 Africa     1997    52.2 14195809     1694.
 8 Central African Republic Africa     1997    46.1  3696513      741.
 9 Chad                     Africa     1997    51.6  7562011     1005.
10 Comoros                  Africa     1997    60.7   527982     1174.
# … with 42 more rows
```

filter() with Conditionals II

```r
# look only at African observations OR observations in 1997
gapminder %>%
  filter(continent == "Africa" |
           year == 1997)
```

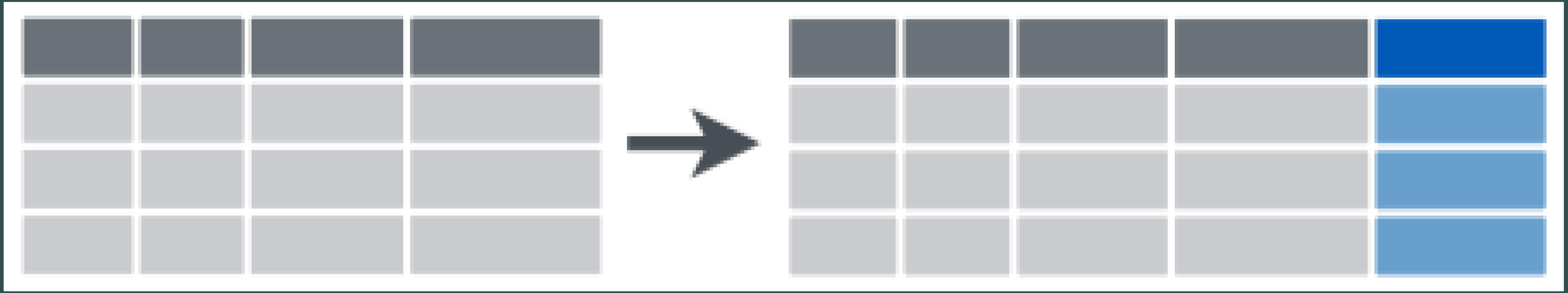
```
# A tibble: 714 × 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1997    41.8 22227415      635.
 2 Albania     Europe     1997    73.0  3428038     3193.
 3 Algeria     Africa     1952    43.1  9279525     2449.
 4 Algeria     Africa     1957    45.7 10270856     3014.
 5 Algeria     Africa     1962    48.3 11000948     2551.
 6 Algeria     Africa     1967    51.4 12760499     3247.
 7 Algeria     Africa     1972    54.5 14760787     4183.
 8 Algeria     Africa     1977    58.0 17152804     4910.
 9 Algeria     Africa     1982    61.4 20033753     5745.
10 Algeria     Africa     1987    65.8 23254956     5681.
# … with 704 more rows
```

filter() with Conditionals III

```
# look only at U.S. and U.K. observations in 2002
gapminder %>%
  filter(country %in%
           c("United States",
             "United Kingdom"),
         year == 2002)
```

```
# A tibble: 2 × 6
  country        continent  year lifeExp       pop gdpPercap
  <fct>          <fct>     <int>   <dbl>     <int>     <dbl>
1 United Kingdom Europe     2002    78.5  59912431    29479.
2 United States  Americas   2002    77.3 287675526    39097.
```

`mutate()`: Create New Variables



`mutate()`

- `mutate` creates a new variable (column)
  - always adds a new column at the end
  - general formula: `new_variable_name = operation`
- Three major types of mutates:

1. Create a variable that is a specific value (often categorical)
2. Change an existing variable (often rescaling)
3. Create a variable based on other variables

mutate(): Setting a Specific Value

```r
# create variable called "europe" if country is in Europe
mutate(gapminder,
       europe = case_when(continent == "Europe" ~ "In Europe",
                          continent != "Europe" ~ "Not in Europe"))
```

```
# A tibble: 1,704 × 7
   country     continent  year lifeExp      pop gdpPercap europe
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl> <chr>
 1 Afghanistan Asia       1952    28.8  8425333      779. Not in Europe
 2 Afghanistan Asia       1957    30.3  9240934      821. Not in Europe
 3 Afghanistan Asia       1962    32.0 10267083      853. Not in Europe
 4 Afghanistan Asia       1967    34.0 11537966      836. Not in Europe
 5 Afghanistan Asia       1972    36.1 13079460      740. Not in Europe
 6 Afghanistan Asia       1977    38.4 14880372      786. Not in Europe
 7 Afghanistan Asia       1982    39.9 12881816      978. Not in Europe
 8 Afghanistan Asia       1987    40.8 13867957      852. Not in Europe
 9 Afghanistan Asia       1992    41.7 16317921      649. Not in Europe
10 Afghanistan Asia       1997    41.8 22227415      635. Not in Europe
# … with 1,694 more rows
```

mutate(): Changing a Variable's Scale

```r
# create population in millions variable
gapminder %>%
  mutate(pop_mil = pop / 1000000)
```

```
# A tibble: 1,704 × 7
   country     continent  year lifeExp      pop gdpPercap pop_mil
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>   <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.    8.43
 2 Afghanistan Asia       1957    30.3  9240934      821.    9.24
 3 Afghanistan Asia       1962    32.0 10267083      853.   10.3
 4 Afghanistan Asia       1967    34.0 11537966      836.   11.5
 5 Afghanistan Asia       1972    36.1 13079460      740.   13.1
 6 Afghanistan Asia       1977    38.4 14880372      786.   14.9
 7 Afghanistan Asia       1982    39.9 12881816      978.   12.9
 8 Afghanistan Asia       1987    40.8 13867957      852.   13.9
 9 Afghanistan Asia       1992    41.7 16317921      649.   16.3
10 Afghanistan Asia       1997    41.8 22227415      635.   22.2
# … with 1,694 more rows
```

mutate(): Variable Based on Other Variables

```r
# create GDP variable from gdpPercap and pop, in billions
gapminder %>%
  mutate(GDP = ((gdpPercap * pop) / 1000000000))
```

```
# A tibble: 1,704 × 7
   country     continent  year lifeExp      pop gdpPercap   GDP
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl> <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.  6.57
 2 Afghanistan Asia       1957    30.3  9240934      821.  7.59
 3 Afghanistan Asia       1962    32.0 10267083      853.  8.76
 4 Afghanistan Asia       1967    34.0 11537966      836.  9.65
 5 Afghanistan Asia       1972    36.1 13079460      740.  9.68
 6 Afghanistan Asia       1977    38.4 14880372      786. 11.7
 7 Afghanistan Asia       1982    39.9 12881816      978. 12.6
 8 Afghanistan Asia       1987    40.8 13867957      852. 11.8
 9 Afghanistan Asia       1992    41.7 16317921      649. 10.6
10 Afghanistan Asia       1997    41.8 22227415      635. 14.1
# … with 1,694 more rows
```

`mutate()`: Change Class of Variable

- Change `class` of a variable inside `mutate()` with `as.*()`

```r
# change year variable from an integer to a factor
gapminder %>%
  mutate(year = as.factor(year))
```

```
# A tibble: 1,704 × 6
   country     continent year  lifeExp      pop gdpPercap
   <fct>       <fct>     <fct>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia      1952     28.8  8425333      779.
 2 Afghanistan Asia      1957     30.3  9240934      821.
 3 Afghanistan Asia      1962     32.0 10267083      853.
 4 Afghanistan Asia      1967     34.0 11537966      836.
 5 Afghanistan Asia      1972     36.1 13079460      740.
 6 Afghanistan Asia      1977     38.4 14880372      786.
 7 Afghanistan Asia      1982     39.9 12881816      978.
 8 Afghanistan Asia      1987     40.8 13867957      852.
 9 Afghanistan Asia      1992     41.7 16317921      649.
10 Afghanistan Asia      1997     41.8 22227415      635.
# … with 1,694 more rows
```

mutate(): Create Multiple Variables

- Can create multiple new variables with commas:

```
gapminder %>%
  mutate(GDP = gdpPercap * pop,
         pop_millions = pop / 1000000)
```

```
# A tibble: 1,704 × 8
   country     continent  year lifeExp      pop gdpPercap          GDP pop_mil…¹
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.  6567086330.      8.43
 2 Afghanistan Asia       1957    30.3  9240934      821.  7585448670.      9.24
 3 Afghanistan Asia       1962    32.0 10267083      853.  8758855797.     10.3
 4 Afghanistan Asia       1967    34.0 11537966      836.  9648014150.     11.5
 5 Afghanistan Asia       1972    36.1 13079460      740.  9678553274.     13.1
 6 Afghanistan Asia       1977    38.4 14880372      786. 11697659231.     14.9
 7 Afghanistan Asia       1982    39.9 12881816      978. 12598563401.     12.9
 8 Afghanistan Asia       1987    40.8 13867957      852. 11820990309.     13.9
 9 Afghanistan Asia       1992    41.7 16317921      649. 10595901589.     16.3
10 Afghanistan Asia       1997    41.8 22227415      635. 14121995875.     22.2
# … with 1,694 more rows, and abbreviated variable name ¹pop_millions
```

`transmute()`: Keep Only New Variables

- `transmute` keeps *only* newly created variables (it `select()`s only the new `mutate()`d variables)

```
gapminder %>%
  transmute(GDP = gdpPercap * pop,
            pop_millions = pop / 1000000)
```

```
# A tibble: 1,704 × 2
           GDP pop_millions
         <dbl>        <dbl>
 1  6567086330.         8.43
 2  7585448670.         9.24
 3  8758855797.        10.3
 4  9648014150.        11.5
 5  9678553274.        13.1
 6 11697659231.        14.9
 7 12598563401.        12.9
 8 11820990309.        13.9
 9 10595901589.        16.3
10 14121995875.        22.2
# … with 1,694 more rows
```

mutate(): Conditionals

- Boolean, logical, and conditionals all work well in `mutate()`:

```r
gapminder %>%
  select(country, year, lifeExp) %>%
  mutate(long_life_1 = lifeExp > 70,
         long_life_2 = case_when(lifeExp > 70 ~ "Long",
                                 lifeExp <= 70 ~ "Short"))
```

```
# A tibble: 1,704 × 5
   country      year lifeExp long_life_1 long_life_2
   <fct>       <int>   <dbl> <lgl>       <chr>
 1 Afghanistan  1952    28.8 FALSE       Short
 2 Afghanistan  1957    30.3 FALSE       Short
 3 Afghanistan  1962    32.0 FALSE       Short
 4 Afghanistan  1967    34.0 FALSE       Short
 5 Afghanistan  1972    36.1 FALSE       Short
 6 Afghanistan  1977    38.4 FALSE       Short
 7 Afghanistan  1982    39.9 FALSE       Short
 8 Afghanistan  1987    40.8 FALSE       Short
 9 Afghanistan  1992    41.7 FALSE       Short
10 Afghanistan  1997    41.8 FALSE       Short
# … with 1,694 more rows
```

`mutate()` is Order Aware

- `mutate()` is order-aware, so you can chain multiple mutates that depend on previous mutates

```r
gapminder %>%
  select(country, year, lifeExp) %>%
  mutate(dog_years = lifeExp * 7,
         comment = paste("Life expectancy in", country, "is", dog_years, "in dog years.", sep = " "))
```

```
# A tibble: 1,704 × 5
   country      year lifeExp dog_years comment
   <fct>       <int>   <dbl>     <dbl> <chr>
 1 Afghanistan  1952    28.8      202. Life expectancy in Afghanistan is 201.60…
 2 Afghanistan  1957    30.3      212. Life expectancy in Afghanistan is 212.32…
 3 Afghanistan  1962    32.0      224. Life expectancy in Afghanistan is 223.97…
 4 Afghanistan  1967    34.0      238. Life expectancy in Afghanistan is 238.14…
 5 Afghanistan  1972    36.1      253. Life expectancy in Afghanistan is 252.61…
 6 Afghanistan  1977    38.4      269. Life expectancy in Afghanistan is 269.06…
 7 Afghanistan  1982    39.9      279. Life expectancy in Afghanistan is 278.97…
 8 Afghanistan  1987    40.8      286. Life expectancy in Afghanistan is 285.75…
 9 Afghanistan  1992    41.7      292. Life expectancy in Afghanistan is 291.71…
10 Afghanistan  1997    41.8      292. Life expectancy in Afghanistan is 292.34…
# … with 1,694 more rows
```

`mutate()`: Scoped-functions I

- "Scoped" variants of `mutate` that work on a subset of variables:
  - `mutate_all()` affects every variable
  - `mutate_at()` affects named or selected variables
  - `mutate_if()` affects variables that meet a criteria

```r
# round all observations of numeric variables to 2 digits
gapminder %>%
  mutate_if(is.numeric, round, digits = 2)
```

```
# A tibble: 1,704 × 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32   10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.8 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

`mutate()`: Scoped-functions II

- "Scoped" variants of `mutate` that work on a subset of variables:
  - `mutate_all()` affects every variable
  - `mutate_at()` affects named or selected variables
  - `mutate_if()` affects variables that meet a criteria

```
# make all factor variables uppercase
gapminder %>%
  mutate_if(is.factor, toupper)
```

```
# A tibble: 1,704 × 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <int>   <dbl>    <int>     <dbl>
 1 AFGHANISTAN ASIA       1952    28.8  8425333      779.
 2 AFGHANISTAN ASIA       1957    30.3  9240934      821.
 3 AFGHANISTAN ASIA       1962    32.0 10267083      853.
 4 AFGHANISTAN ASIA       1967    34.0 11537966      836.
 5 AFGHANISTAN ASIA       1972    36.1 13079460      740.
 6 AFGHANISTAN ASIA       1977    38.4 14880372      786.
 7 AFGHANISTAN ASIA       1982    39.9 12881816      978.
 8 AFGHANISTAN ASIA       1987    40.8 13867957      852.
 9 AFGHANISTAN ASIA       1992    41.7 16317921      649.
10 AFGHANISTAN ASIA       1997    41.8 22227415      635.
# … with 1,694 more rows
```

A Reminder on Viewing, Saving, & Overwriting Objects I

- `dplyr` functions never modify their inputs (i.e. never overwrite the original `tibble`)
- If you want to save a result, use `<-` to assign it to a new `tibble`
- If assigned, you will not see the output until you call up the new `tibble` by name

```
# Prints output, doesn't save/overwrite object
gapminder %>%
  filter(continent == "Africa")
```

```
# A tibble: 624 × 6
   country continent  year lifeExp      pop gdpPercap
   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
 1 Algeria Africa     1952    43.1  9279525     2449.
 2 Algeria Africa     1957    45.7 10270856     3014.
 3 Algeria Africa     1962    48.3 11000948     2551.
 4 Algeria Africa     1967    51.4 12760499     3247.
 5 Algeria Africa     1972    54.5 14760787     4183.
 6 Algeria Africa     1977    58.0 17152804     4910.
 7 Algeria Africa     1982    61.4 20033753     5745.
 8 Algeria Africa     1987    65.8 23254956     5681.
 9 Algeria Africa     1992    67.7 26298373     5023.
10 Algeria Africa     1997    69.2 29072015     4797.
# … with 614 more rows
```

```
# Saves as africa
africa <- gapminder %>%
  filter(continent == "Africa")
```

```r

# Look at it
africa
```

```
# A tibble: 624 × 6
   country continent  year lifeExp      pop gdpPercap
   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
 1 Algeria Africa     1952    43.1  9279525     2449.
 2 Algeria Africa     1957    45.7 10270856     3014.
 3 Algeria Africa     1962    48.3 11000948     2551.
 4 Algeria Africa     1967    51.4 12760499     3247.
 5 Algeria Africa     1972    54.5 14760787     4183.
 6 Algeria Africa     1977    58.0 17152804     4910.
 7 Algeria Africa     1982    61.4 20033753     5745.
 8 Algeria Africa     1987    65.8 23254956     5681.
 9 Algeria Africa     1992    67.7 26298373     5023.
10 Algeria Africa     1997    69.2 29072015     4797.
# … with 614 more rows
```

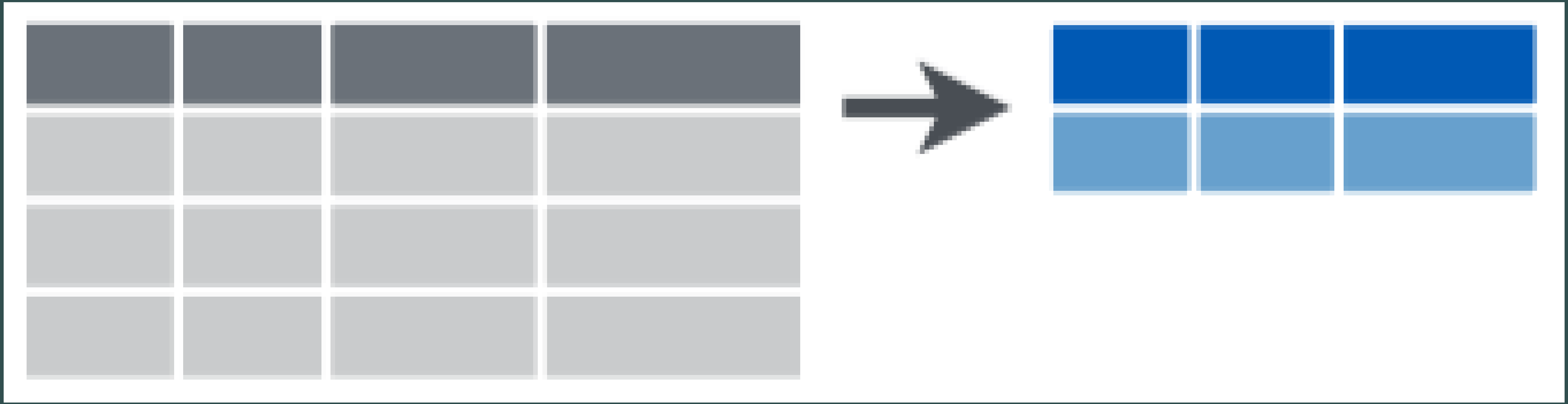
A Reminder on Viewing, Saving, & Overwriting Objects II

- Neat trick:

```
# Save and view at same time by wrapping whole command with ()
(africa <- gapminder %>%
  filter(continent == "Africa"))
```

```
# A tibble: 624 × 6
   country continent  year lifeExp      pop gdpPercap
   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
 1 Algeria Africa     1952    43.1  9279525     2449.
 2 Algeria Africa     1957    45.7 10270856     3014.
 3 Algeria Africa     1962    48.3 11000948     2551.
 4 Algeria Africa     1967    51.4 12760499     3247.
 5 Algeria Africa     1972    54.5 14760787     4183.
 6 Algeria Africa     1977    58.0 17152804     4910.
 7 Algeria Africa     1982    61.4 20033753     5745.
 8 Algeria Africa     1987    65.8 23254956     5681.
 9 Algeria Africa     1992    67.7 26298373     5023.
10 Algeria Africa     1997    69.2 29072015     4797.
# … with 614 more rows
```

`summarize()`: Create Statistics



summarize()

- summarize[1] outputs a tibble of desired summary statistics
  - can name the statistic variable as if you were mutate()-ing a new variable

```
# get average life expectancy and call it avg_LE

gapminder %>%
  summarize(avg_LE = mean(lifeExp))
```

```
# A tibble: 1 × 1
  avg_LE
   <dbl>
1   59.5
```


1. Also the more civilised non-U.S. English spelling summarise also works. dplyr was written by a Kiwi after all!

`summarize()`: Useful commands

- Useful `summarize()` commands:

Command	Does
`n()`	Number of observations
`n_distinct()`	Number of unique observations
`sum()`	Sum all observations of a variable
`mean()`	Average of all observations of a variable
`median()`	50^{th} percentile of all observations of a variable
`sd()`	Standard deviation of all observations of a variable

Most commands require you to put a variable name inside the command's argument parentheses. `n()` and `n_distinct()` require empty parentheses!

`summarize()`: Useful commands II

- Useful `summarize()` commands:

Command	Does
`min()`	Minimum value of a variable
`max()`	Maximum value of a variable
`quantile(., 0.25)`	Specified percentile (e.g. 25th percentile) of a variable
`first()`	First value of a variable
`last()`	Last value of a variable
`nth(., 2)`	Specified position of a variable (example 2nd)

The `.` in `quantile()` and `nth()` are where you would put your variable name.

summarize() counts

- Counts of a categorical variable are useful, and can be done a few different ways:

```r
# summarize with n() gives size of current group, has no arguments
gapminder %>%
  summarize(amount = n()) # I've called it "amount"
```

```
# A tibble: 1 × 1
  amount
   <int>
1   1704
```

```r
# count() is a dedicated command, counts observations by specified variable
gapminder %>%
  count(year) # counts how many observations per year
```

```
# A tibble: 12 × 2
    year     n
   <int> <int>
 1  1952   142
 2  1957   142
 3  1962   142
 4  1967   142
 5  1972   142
 6  1977   142
 7  1982   142
 8  1987   142
 9  1992   142
10  1997   142
```

`summarize()` Conditionally

- Can do counts and proportions by conditions
  - How many observations fit specified conditions (e.g. `TRUE`)
  - Numeric objects: `TRUE=1` and `FALSE=0`
    - `sum(x)` becomes the number of `TRUE`s in `x`
    - `mean(x)` becomes the proportion

```r
# How many countries have life
# expectancy over 70 in 2007?
gapminder %>%
  filter(year=="2007") %>%
  summarize(Over_70 = sum(lifeExp>70))
```

```
# A tibble: 1 × 1
  Over_70
    <int>
1      83
```

```r
# What *proportion* of countries have life
# expectancy over 70 in 2007?
gapminder %>%
  filter(year=="2007") %>%
  summarize(Over_70 = mean(lifeExp>70))
```

```
# A tibble: 1 × 1
  Over_70
    <dbl>
1   0.585
```

`summarize()` Multiple Variables

- Can `summarize()` multiple *variables* at once, separate by commas

```
# get average life expectancy and GDP
# call each avg_LE, avg_GDP
gapminder %>%
  summarize(avg_LE = mean(lifeExp),
            avg_GDP = mean(gdpPercap))
```

```
# A tibble: 1 × 2
  avg_LE avg_GDP
   <dbl>   <dbl>
1   59.5   7215.
```

`summarize()` Multiple Statistics

- Can `summarize()` multiple *statistics* of a variable at once, separate by commas

```
# get count, mean, sd, min, max
# of life Expectancy
gapminder %>%
  summarize(obs = n(),
            avg_LE = mean(lifeExp),
            sd_LE = sd(lifeExp),
            min_LE = min(lifeExp),
            max_LE = max(lifeExp))
```

```
# A tibble: 1 × 5
    obs avg_LE sd_LE min_LE max_LE
  <int>  <dbl> <dbl>  <dbl>  <dbl>
1  1704   59.5  12.9   23.6   82.6
```

`summarize()` Scoped Versions

- "Scoped" versions of `summarize()` that work on a subset of variables
  - `summarize_all()`: affects every variable
  - `summarize_at()`: affects named or selected variables
  - `summarize_if()`: affects variables that meet a criteria

```r
# get the average of all
# numeric variables
gapminder %>%
  summarize_if(is.numeric,
               funs(avg = mean))
```

```
# A tibble: 1 × 4
  year_avg lifeExp_avg   pop_avg gdpPercap_avg
     <dbl>       <dbl>     <dbl>         <dbl>
1    1980.        59.5 29601212.         7215.
```

```r
# get mean and sd for
# pop and lifeExp

gapminder %>%
  summarize_at(vars(pop, lifeExp),
    funs("avg" = mean,
         "std dev" = sd))
```

```
# A tibble: 1 × 4
    pop_avg lifeExp_avg `pop_std dev` `lifeExp_std
dev`
      <dbl>       <dbl>         <dbl>
<dbl>
1 29601212.        59.5    106157897.
12.9
```

`group_by()`: Grouped summaries

`group_by()` + `summarize()` I

- If we have `factor` variables grouping a variable into categories, we can run `dplyr` verbs by group
  - Particularly useful for `summarize()`
- First define the group with `group_by()`

```r
# get average life expectancy and gdp by continent
gapminder %>%
  group_by(continent) %>%
  summarize(avg_life = mean(lifeExp),
            avg_GDP = mean(gdpPercap))
```

```
# A tibble: 5 × 3
  continent avg_life avg_GDP
  <fct>        <dbl>   <dbl>
1 Africa        48.9   2194.
2 Americas      64.7   7136.
3 Asia          60.1   7902.
4 Europe        71.9  14469.
5 Oceania       74.3  18622.
```

`group_by()` + `summarize()` II

```r
# track changes in average life expectancy and gdp over time
gapminder %>%
  group_by(year) %>%
  summarize(mean_life = mean(lifeExp),
            mean_GDP = mean(gdpPercap))
```

```
# A tibble: 12 × 3
    year mean_life mean_GDP
   <int>     <dbl>    <dbl>
 1  1952      49.1    3725.
 2  1957      51.5    4299.
 3  1962      53.6    4726.
 4  1967      55.7    5484.
 5  1972      57.6    6770.
 6  1977      59.6    7313.
 7  1982      61.5    7519.
 8  1987      63.2    7901.
 9  1992      64.2    8159.
10  1997      65.0    9090.
11  2002      65.7    9918.
```

group_by() + summarize() III

- Can group observations by multiple variables (in proper order)

```r
# track changes in average life expectancy and gdp over time
gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_life = mean(lifeExp),
            mean_GDP = mean(gdpPercap))
```

```
# A tibble: 60 × 4
# Groups:   continent [5]
   continent  year mean_life mean_GDP
   <fct>     <int>     <dbl>    <dbl>
 1 Africa     1952      39.1    1253.
 2 Africa     1957      41.3    1385.
 3 Africa     1962      43.3    1598.
 4 Africa     1967      45.3    2050.
 5 Africa     1972      47.5    2340.
 6 Africa     1977      49.6    2586.
 7 Africa     1982      51.6    2482.
 8 Africa     1987      53.3    2283.
 9 Africa     1992      53.6    2282.
10 Africa     1997      53.6    2379.
```

Piping Across Packages

- `tidyverse` uses same grammar and design philosophy

Code | Output

```
gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_life = mean(lifeExp),
            mean_GDP = mean(gdpPercap)) %>%
  # now pipe this tibble in as data for ggplot!
  ggplot(data = ., # . pipes the above in (to data layer)
         aes(x = year,
             y = mean_life,
             color = continent))+
  geom_path(size = 1)+
  labs(x = "Year",
       y = "Average Life Expectancy (Years)",
       color = "Continent",
       title = "Average Life Expectancy Over Time")+
  theme_classic(base_family = "Fira Sans Condensed",
                base_size = 20)
```

`dplyr`: Other Useful Commands

`tally()`: counts for categories

- `tally` provides counts, best used with `group_by` for `factors`

```r
gapminder %>%
  tally
```

```
# A tibble: 1 × 1
      n
  <int>
1  1704
```

```r
gapminder %>%
  group_by(continent) %>%
  tally
```

```
# A tibble: 5 × 2
  continent     n
  <fct>     <int>
1 Africa      624
2 Americas    300
3 Asia        396
4 Europe      360
5 Oceania      24
```

`slice()`: Filter row by *position*

- `slice()` subsets observations by *position* instead of `filter`ing by *values*

```
gapminder %>%
  slice(15:17) # see 15th through 17th rows
```

```
# A tibble: 3 × 6
  country continent  year lifeExp     pop gdpPercap
  <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
1 Albania Europe     1962    64.8 1728137     2313.
2 Albania Europe     1967    66.2 1984060     2760.
3 Albania Europe     1972    67.7 2263554     3313.
```

```
gapminder %>%
  slice(c(2,3,150)) # see 2nd, 3rd, and 150th rows
```

```
# A tibble: 3 × 6
  country                continent  year lifeExp      pop gdpPercap
  <fct>                  <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan            Asia       1957    30.3  9240934      821.
2 Afghanistan            Asia       1962    32.0 10267083      853.
3 Bosnia and Herzegovina Europe     1977    69.9  4086000     3528.
```

`pull()`: Extract columns

- `pull()` extracts a column from a `tibble` (just like `$` for a `data.frame`)

```
# Get all U.S. life expectancy observations
gapminder %>%
  filter(country == "United States") %>%
  pull(lifeExp)
```

```
 [1] 68.440 69.490 70.210 70.760 71.340 73.380 74.650 75.020 76.090 76.810
[11] 77.310 78.242
```

```
# Note this is basically a vector!
```

```
# Get U.S. life expectancy in 2007
gapminder %>%
  filter(country == "United States" & year == 2007) %>%
  pull(lifeExp)
```

```
[1] 78.242
```

```
# Here's just one value now
```

- Good for extracting & saving important values as objects for further use

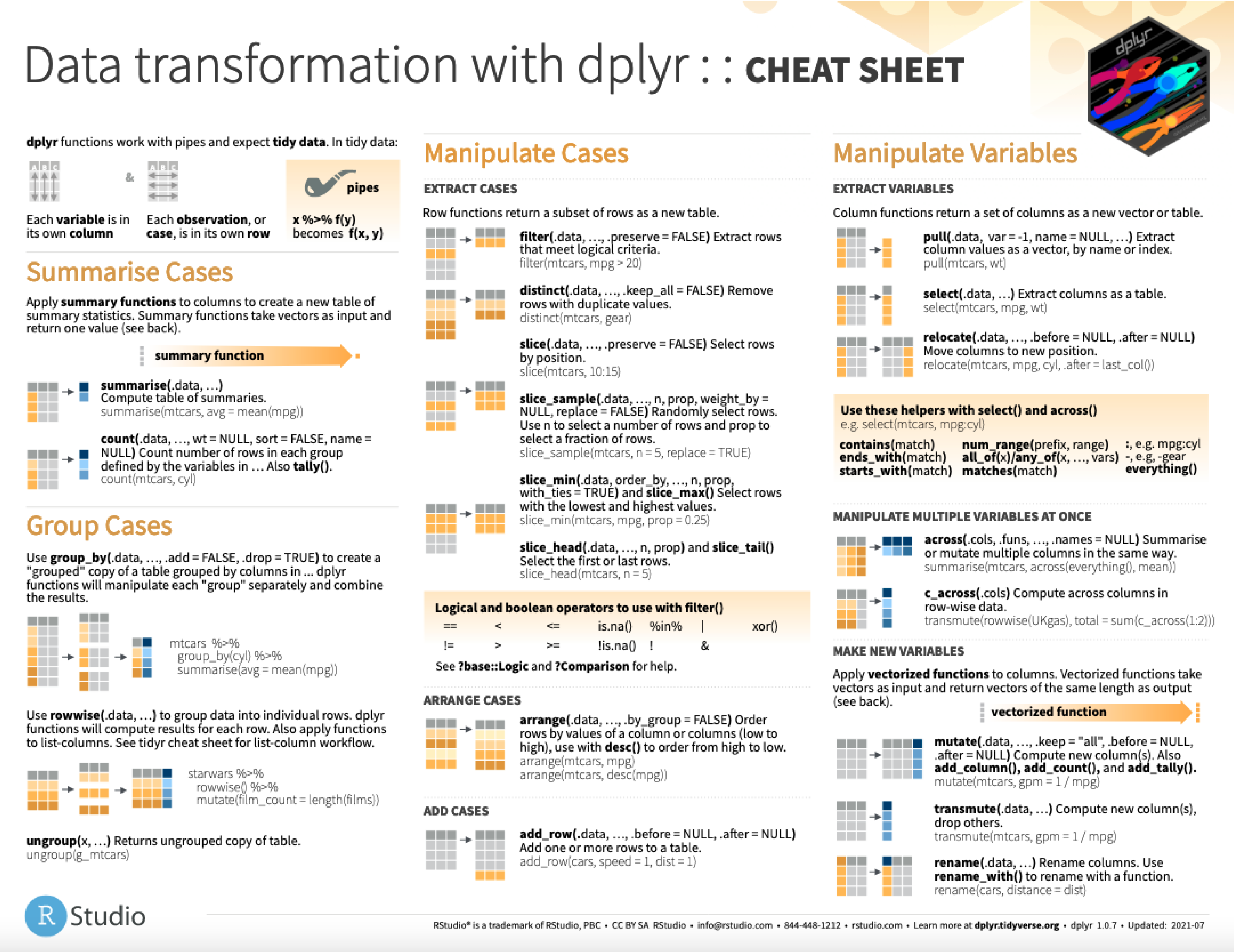
`distinct()`: Show unique values

- `distinct()` shows the distinct values of a specified variable (recall `n_distinct()` inside `summarize()` just gives you the *number* of values)

```
gapminder %>%
  distinct(country)
```

```
# A tibble: 142 × 1
   country
   <fct>
 1 Afghanistan
 2 Albania
 3 Algeria
 4 Angola
 5 Argentina
 6 Australia
 7 Austria
 8 Bahrain
 9 Bangladesh
10 Belgium
# … with 132 more rows
```

Data Wrangling Cheat Sheet



Data transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes: x %>% f(y) becomes f(x, y)

Summarise Cases

Apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, ...)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(.data, ..., wt = NULL, sort = FALSE, name = NULL) Count number of rows in each group defined by the variables in ... Also **tally()**.
count(mtcars, cyl)

Group Cases

Use **group_by**(.data, ..., .add = FALSE, .drop = TRUE) to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.

```
mtcars %>%
  group_by(cyl) %>%
  summarise(avg = mean(mpg))
```

Use **rowwise**(.data, ...) to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyr cheat sheet for list-column workflow.

```
starwars %>%
  rowwise() %>%
  mutate(film_count = length(films))
```

ungroup(x, ...) Returns ungrouped copy of table.
ungroup(g_mtcars)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, ..., .preserve = FALSE) Extract rows that meet logical criteria.
filter(mtcars, mpg > 20)

distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
distinct(mtcars, gear)

slice(.data, ..., .preserve = FALSE) Select rows by position.
slice(mtcars, 10:15)

slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE) Randomly select rows. Use n to select a number of rows and prop to select a fraction of rows.
slice_sample(mtcars, n = 5, replace = TRUE)

slice_min(.data, order_by, ..., n, prop, with_ties = TRUE) and **slice_max()** Select rows with the lowest and highest values.
slice_min(mtcars, mpg, prop = 0.25)

slice_head(.data, ..., n, prop) and **slice_tail()** Select the first or last rows.
slice_head(mtcars, n = 5)

Logical and boolean operators to use with filter()

==	<	<=	is.na()	%in%	|	xor()
!=	>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, ..., .by_group = FALSE) Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES

add_row(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table.
add_row(cars, speed = 1, dist = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1, name = NULL, ...) Extract column values as a vector, by name or index.
pull(mtcars, wt)

select(.data, ...) Extract columns as a table.
select(mtcars, mpg, wt)

relocate(.data, ..., .before = NULL, .after = NULL) Move columns to new position.
relocate(mtcars, mpg, cyl, .after = last_col())

Use these helpers with select() and across()
e.g. select(mtcars, mpg:cyl)

contains(match) **num_range**(prefix, range) :, e.g. mpg:cyl
ends_with(match) **all_of**(x)/**any_of**(x, ..., vars) -, e.g, -gear
starts_with(match) **matches**(match) **everything()**

MANIPULATE MULTIPLE VARIABLES AT ONCE

across(.cols, .funs, ..., .names = NULL) Summarise or mutate multiple columns in the same way.
summarise(mtcars, across(everything(), mean))

c_across(.cols) Compute across columns in row-wise data.
transmute(rowwise(UKgas), total = sum(c_across(1:2)))

MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).

vectorized function

mutate(.data, ..., .keep = "all", .before = NULL, .after = NULL) Compute new column(s). Also **add_column()**, **add_count()**, and **add_tally()**.
mutate(mtcars, gpm = 1 / mpg)

transmute(.data, ...) Compute new column(s), drop others.
transmute(mtcars, gpm = 1 / mpg)

rename(.data, ...) Rename columns. Use **rename_with()** to rename with a function.
rename(cars, distance = dist)

RStudio

R Studio: Data Wrangling Cheat Sheet

Resources

- `tibble`
  - *R For Data Science*, Chapter 10: Tibbles
- `readr` and importing data
  - *R For Data Science*, Chapter 11: Data Import
  - R Studio Cheatsheet: Data Import
- `dplyr` and data wrangling
  - *R For Data Science*, Chapter 5: Data Transformation
  - R Studio Cheatsheet: Data Wrangling (New version)
- `tidyr` and tidying or reshaping data
  - *R For Data Science*, Chapter 12: Tidy Data
  - R Studio Cheatsheet: Data Wrangling
  - R Studio Cheatsheet: Data Import

- Joining data
  - *R For Data Science*, Chapter 13: Relational Data
  - R Studio Cheatsheet: Data Transformation