

Data Wrangling

 zasghar@qau.edu.pk
 [zahedasghar](https://www.zahedasghar.com)
 zahidasghar.com



Contents

Tibbles & Piping

Importing Data

Tidying (Pivoting/Reshaping) Data

Joining Datasets

Wrangling Data

`select()` Variables

`filter()` Select Rows by Condition

`mutate()` Create New Variables

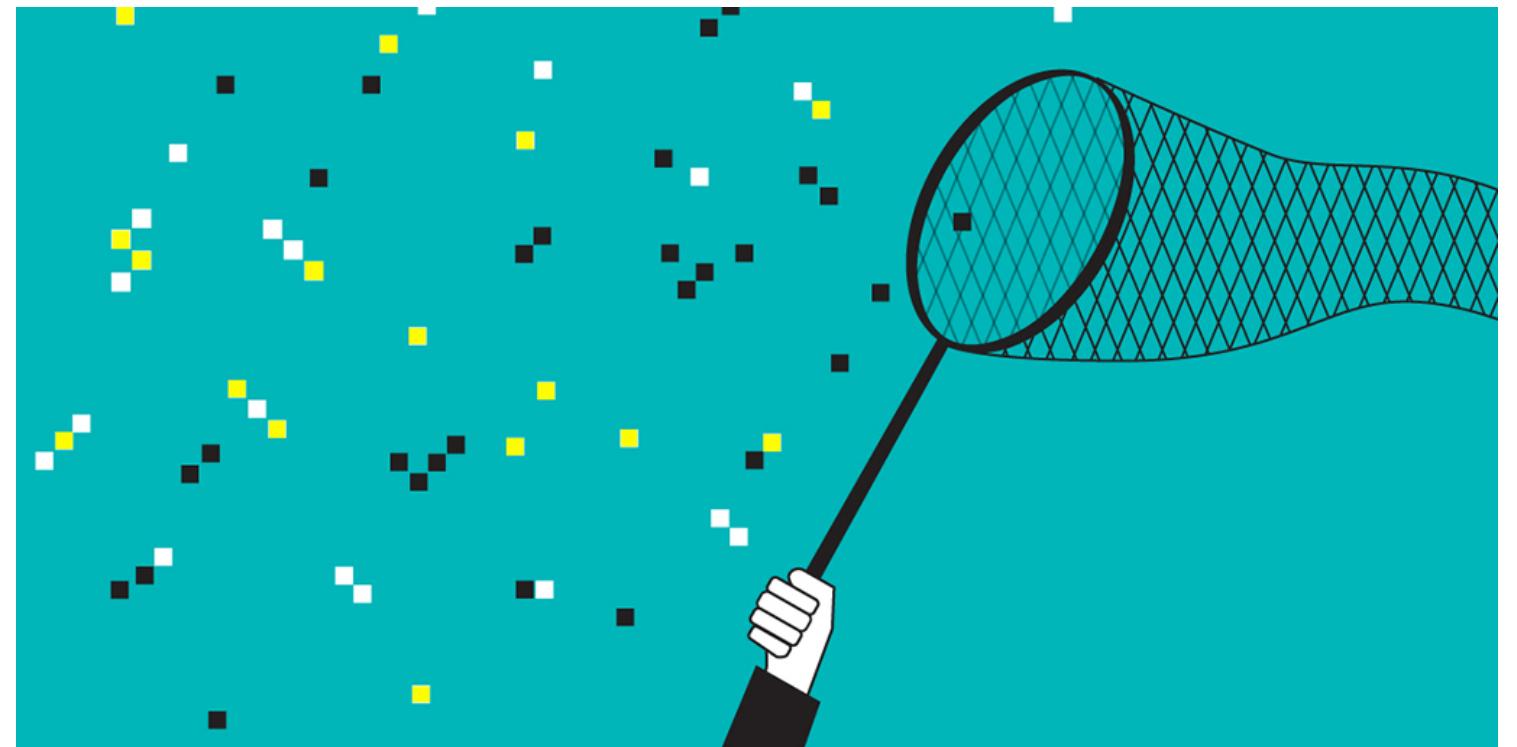
`summarize()` Create Statistics

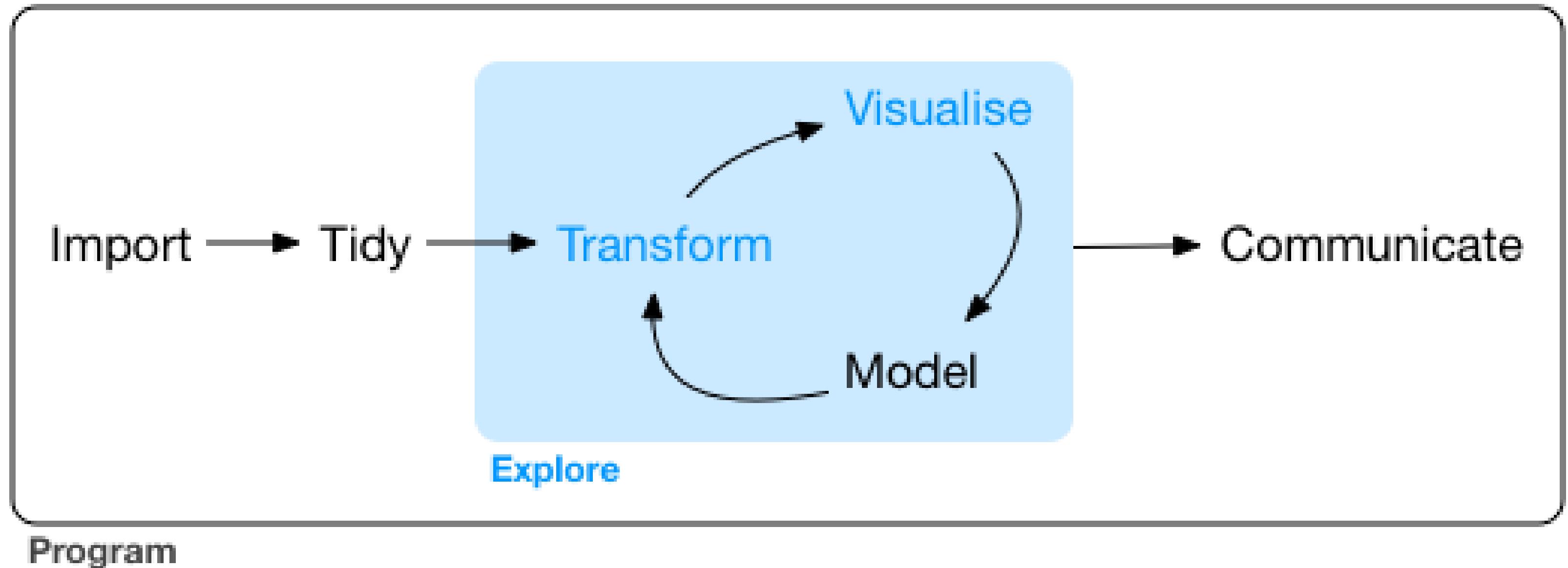
`group_by()` Grouped Summaries

`dplyr` Other Useful Commands

Data Wrangling

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





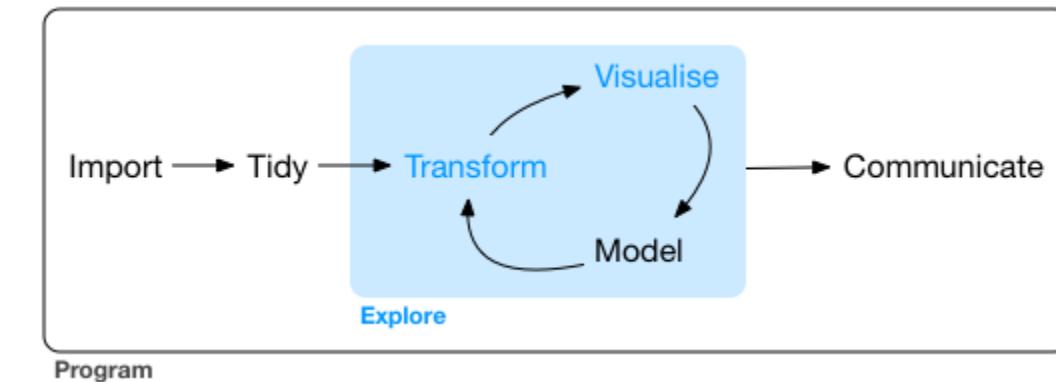
Workflow of a Data Scientist I

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

Zahid Asghar

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 III

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

The tidyverse IV

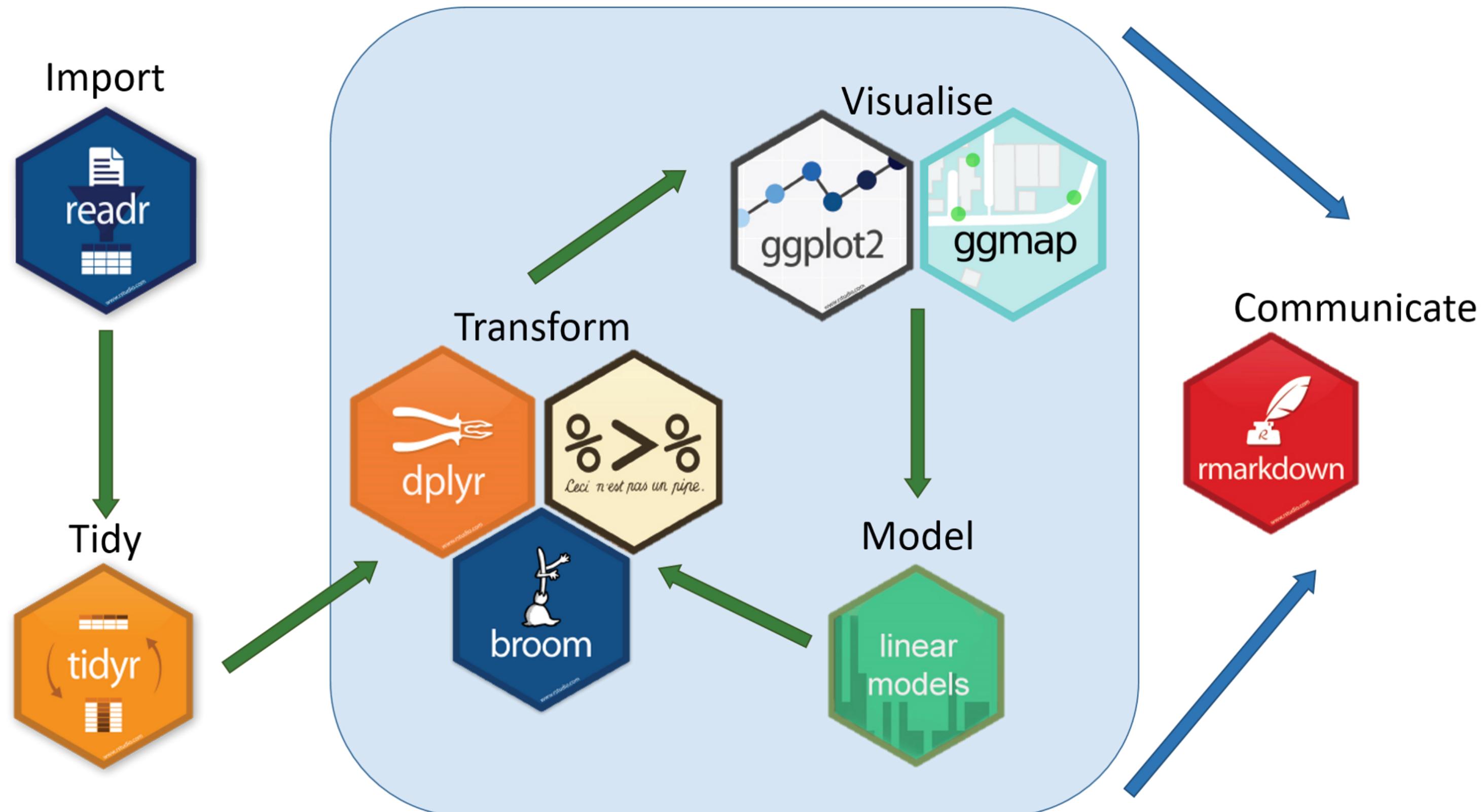
- `tidyverse` contains a lot of packages

```
1 tidyverse_packages()
```

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

- 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

```

1 # look at data
2 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

```
1 # another useful command
2 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, 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()**

```
1 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()`

```
1 example <- tibble(x = seq(2,6,2), # sequence from 2 to 6 by 2's
2                         y = rnorm(3,0,1), # 3 random draws with mean 0, sd 1
3                         colors = c("orange", "green", "blue")) # colors
4
5 example # look at it
# A tibble: 3 × 3
  x     y   colors
  <dbl> <dbl> <chr>
1     2  0.614 orange
2     4 -2.08  green
3     6  0.768 blue
```

Tibbles: Making a Tibble (from Scratch)

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



```

1 example_2 <- tribble(
2   ~x, ~y, ~color, # each variable name must start with ~
3   2, 1.5, "orange",
4   4, 0.2, "green",
5   6, 0.8, "blue") # last element has no comma
6
7 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”

Piping Code

- With math functions, typically read from outside \leftarrow (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 \rightarrow right:

```
1 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:

```
1 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:

```
1 mpg %>%
2   filter(manufacturer == "audi") %>%
3   group_by(model) %>%
4   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 directory¹
- 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”

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:

```
1 # Example
2
3 df <- read_csv("C:/Documents and Settings/Ryan Safner/Downloads/my_data.csv")
```

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

```
1 # Example
2 # If working directory is Documents, but data is in Downloads, like so:
3 #
4 # Ryan Safner/
5 # |
6 # | - Documents/
7 # | - Downloads/
8 # | - Photos/
9 # | - Videos/
10 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



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

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")

A,B,C
1,2,3
4,5,NA

A	B	C
1	2	3
4	5	NA

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")

A;B;C
1;2;3
4;5;NA

A	B	C
1.5	2	3
4.5	5	NA

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

A B C
1 2 3
4 5 NA

A	B	C
1	2	3
4	5	NA

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", file = "file.tsv")

USEFUL READ ARGUMENTS

A	B	C
1	2	3
4	5	NA

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

1	2	3
4	5	NA

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

x	y	z
A	B	C
1	2	3
4	5	NA

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

A	B	C
NA	2	3
4	5	NA

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

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

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)
```

A	B	C
1	2	3

A,B,C
1,2,3
4,5,NA

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:

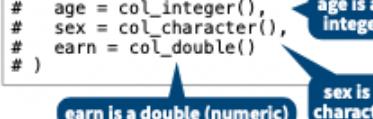
- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

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()
# )
```



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()** - "?"

DEFINE COLUMN SPECIFICATION

Set a default type

```
read_csv(
  file,
  col_type = list(default = col_double())
)
```

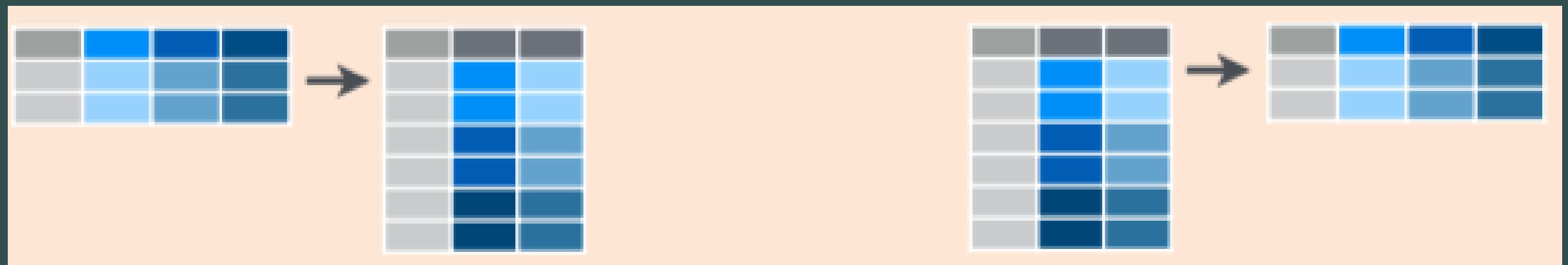
Use column type or string abbreviation

```
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"
)
```

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!

country	year	cases	population
Afghanistan	1999	745	1998071
Afghanistan	2000	2666	2059360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	21366	128042583

variables

country	year	cases	population
Afghanistan	1999	745	1998071
Afghanistan	2000	2666	2059360
Brazil	1999	37737	17200362
Brazil	2000	80488	17450898
China	1999	212258	127291272
China	2000	213766	128042583

observations

Zahid Asghar

country	year	cases	population
Afghanistan	1999	745	1998071
Afghanistan	2000	2666	2059360
Brazil	1999	37737	17200362
Brazil	2000	80488	17450898
China	1999	212258	127291272
China	2000	213766	128042583

values

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

country	year	cases	population
Afghanistan	1990	745	1937071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

variables

country	year	cases	population
Afghanistan	1990	745	1937071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

observations

country	year	cases	population
Afghanistan	1990	745	1937071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

values

Examples of Untidy Data

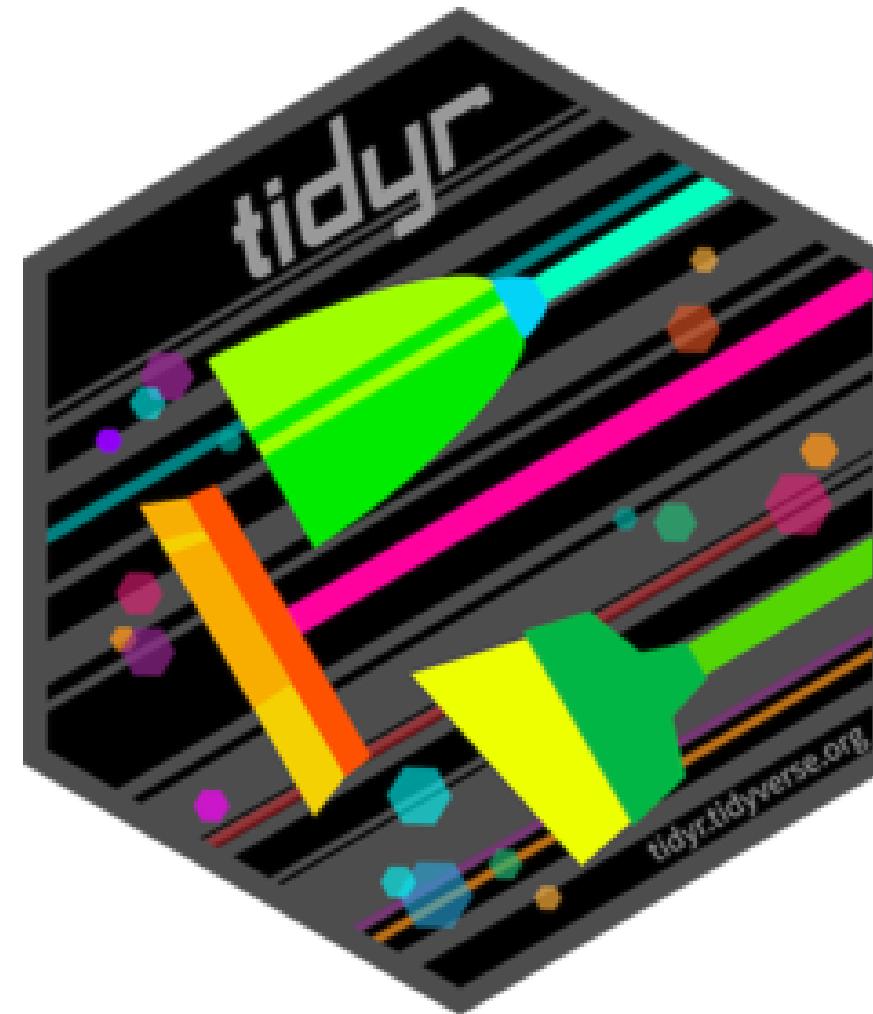
	A	AA	AB	AC	AD	AE	AF	AG	AH
1	Estimated HIV Prevalence% - (Ages 15-49)	2004	2005	2006	2007	2008	2009	2010	2011
2	Abkhazia								
3	Afghanistan						0.06	0.06	0.06
4	Akrotiri and Dhekelia								
5	Albania								
6	Algeria	0.1	0.1	0.1	0.1	0.1			
7	American Samoa								
8	Andorra								
9	Angola	1.9	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

Reshaping/Pivoting Data



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

wide
long

id
key
val

	x	y	z
1	a	c	e
2	b	d	f

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](#)

Zahid Asghar



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

```
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
```

```
1 ex_wide %>%
2   pivot_longer(c("2000","2010"), # select columns
3                 names_to = "year", # variable for column names
4                 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

1 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

```
1 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

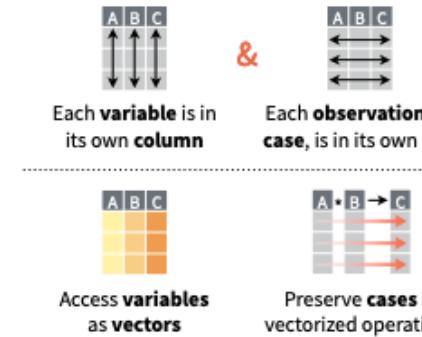
```
1 ex_long
# A tibble: 12 × 4
  country     year type    count
  <chr>       <dbl> <chr>   <dbl>
1 United States 2000 cases     140
2 United States 2000 population 300
3 United States 2010 cases     180
4 United States 2010 population 310
5 Canada        2000 cases     102
6 Canada        2000 population 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
```

```
1 ex_long %>%
2   pivot_wider(names_from = "type", # column with names of var
3               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:



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"))`

`tibble(...)` Construct by rows.

`tibble(~x, ~y, 1, "a", 2, "b", 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.

table4a		
country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

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

`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")`

table2			
country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

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

`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.

table5		
country	century	year
A	19	99
A	20	00
B	19	99
B	20	00

country	year
A	1999
A	2000
B	1999
B	2000

`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 = "")`

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M

`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"))`

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M

`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).

x	x1 x2 x3
A	1 3
B	1 4
B	2 3

`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)`

x	x1 x2 x3
A	1 3
B	1 4
B	2 3

`complete(data, ..., fill = list())` Add missing possible combinations of values of variables listed in ... Fill remaining variables with NA.

`complete(mtcars, cyl, gear, carb)`

x	x1 x2 x3
A	1
B	NA
C	NA
D	3
E	NA

`drop_na(data, ...)` Drop rows containing NA's in ... columns.

`drop_na(x, x2)`

x	x1 x2
A	1
B	NA
C	NA
D	3
E	NA

`fill(data, ..., .direction = "down")` Fill in NA's in ... columns using the next or previous value.</p

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
<code>filter()</code>	Keep only selected <i>observations</i>
<code>select()</code>	Keep only selected <i>variables</i>
<code>arrange()</code>	Reorder rows (e.g. in numerical order)
<code>mutate()</code>	Create new variables
<code>summarize()</code>	Collapse data into summary statistics
<code>group_by()</code>	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

```

1 # order by smallest to largest pop
2 gapminder %>%
3   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

```

1 # order by year, with the smallest to largest pop in each year
2 gapminder %>%
3   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

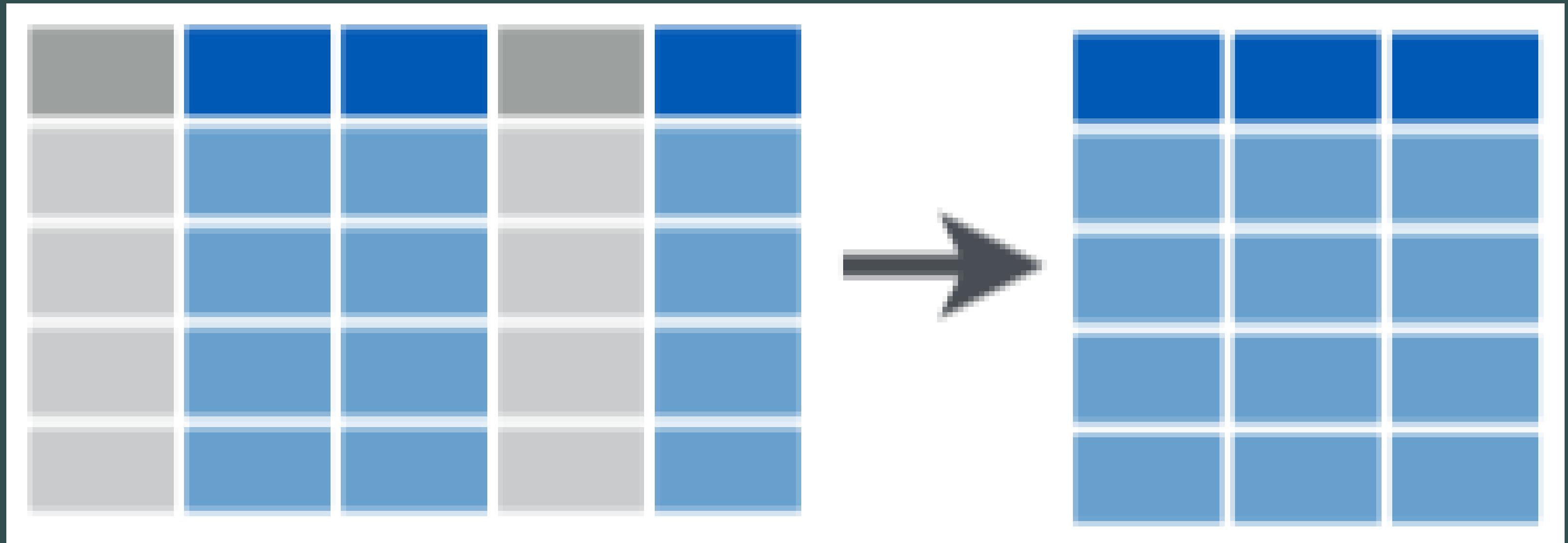
```

arrange(): Descending Order

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

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

select() Variables



select()

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

```
1 # keep only country, year, and population variables
2 gapminder %>%
3   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 -**

```

1 # keep all variables *except* gdpPercap
2 gapminder %>%
3   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

```

1 # move pop to first column
2 gapminder %>%
3   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

```
1 # keep all variables starting with "co"  
2 gapminder %>%  
3   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

```
1 # keep country and all variables containing "per"
2 gapminder %>%
3   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`

```

1 # rename gdpPercap to GDP and lifeExp to population
2 gapminder %>%
3   rename(GDP = gdpPercap,
4         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)

```

1 # look only at African observations
2 gapminder %>%
3   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](#).

filter() with Conditionals I

- Can chain multiple conditions with a `,`

```

1 # look only at African observations in 1997
2 gapminder %>%
3   filter(continent == "Africa",
4         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

```

1 # look only at African observations OR observations in 1997
2 gapminder %>%
3   filter(continent == "Africa" |
4         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

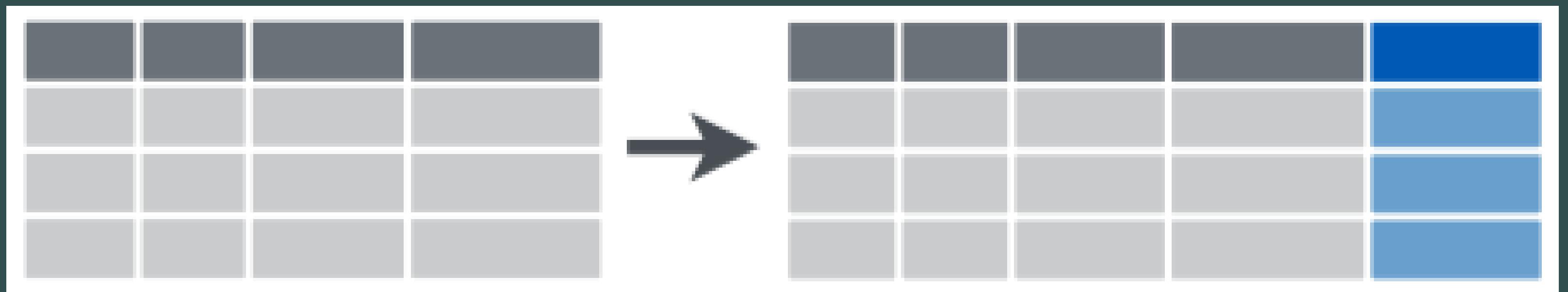
```

filter() with Conditionals III

```
1 # look only at U.S. and U.K. observations in 2002
2 gapminder %>%
3   filter(country %in%
4         c("United States",
5           "United Kingdom"),
6         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

```

1 # create variable called "europe" if country is in Europe
2 mutate(gapminder,
3     europe = case_when(continent == "Europe" ~ "In Europe",
4                         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

```

1 # create population in millions variable
2 gapminder %>%
3   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

```

1 # create GDP variable from gdpPercap and pop, in billions
2 gapminder %>%
3   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.*()`

```

1 # change year variable from an integer to a factor
2 gapminder %>%
3   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:

```

1 gapminder %>%
2   mutate(GDP = gdpPercap * pop,
3         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)

```

1 gapminder %>%
2   transmute(GDP = gdpPercap * pop,
3             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()`:

```

1 gapminder %>%
2   select(country, year, lifeExp) %>%
3   mutate(long_life_1 = lifeExp > 70,
4         long_life_2 = case_when(lifeExp > 70 ~ "Long",
5                                   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

```

1 gapminder %>%
2   select(country, year, lifeExp) %>%
3   mutate(dog_years = lifeExp * 7,
4         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

```
1 # round all observations of numeric variables to 2 digits
2 gapminder %>%
3   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     32.0  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     39.8  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

```

1 # make all factor variables uppercase
2 gapminder %>%
3   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

```
1 # Prints output, doesn't save/overwrite object
2 gapminder %>%
3   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
```

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

```
4  
5 # Look at it  
6 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
```

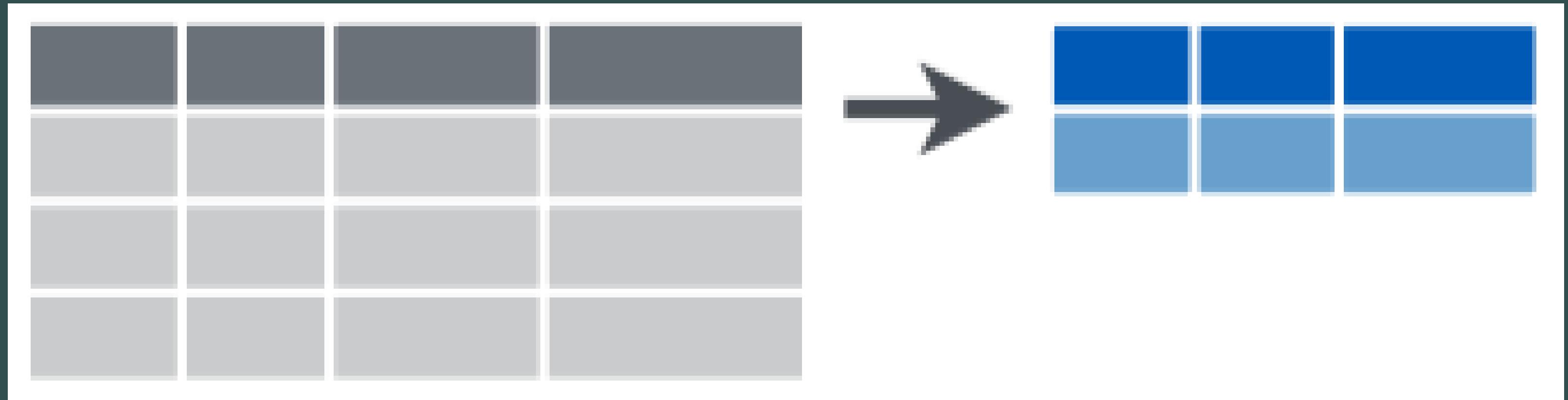
A Reminder on Viewing, Saving, & Overwriting Objects II

- Neat trick:

```
1 # Save and view at same time by wrapping whole command with ()
2 (africa <- gapminder %>%
3   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

```

1 # get average life expectancy and call it avg_LE
2
3 gapminder %>%
4   summarize(avg_LE = mean(lifeExp))

```

```

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

```

summarize(): Useful commands

- Useful `summarize()` commands:

Command	Does
<code>n()</code>	Number of observations
<code>n_distinct()</code>	Number of unique observations
<code>sum()</code>	Sum all observations of a variable
<code>mean()</code>	Average of all observations of a variable
<code>median()</code>	50 th percentile of all observations of a variable
<code>sd()</code>	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. 25 th percentile) of a variable
first()	First value of a variable
last()	Last value of a variable
nth(., 2)	Specified position of a variable (example 2 nd)

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:

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

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

```
1 # count() is a dedicated command, counts observations by specified variable
2 gapminder %>%
3   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
11 2002    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 TRUEs in x
 - `mean(x)` becomes the proportion

```

1 # How many countries have life
2 # expectancy over 70 in 2007?
3 gapminder %>%
4   filter(year=="2007") %>%
5   summarize(Over_70 = sum(lifeExp>70))

```

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

```

1 # What *proportion* of countries have life
2 # expectancy over 70 in 2007?
3 gapminder %>%
4   filter(year=="2007") %>%
5   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

```
1 # get average life expectancy and GDP
2 # call each avg_LE, avg_GDP
3 gapminder %>%
4   summarize(avg_LE = mean(lifeExp),
5             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

```
1 # get count, mean, sd, min, max
2 # of life Expectancy
3 gapminder %>%
4   summarize(obs = n(),
5             avg_LE = mean(lifeExp),
6             sd_LE = sd(lifeExp),
7             min_LE = min(lifeExp),
8             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

```

1 # get the average of all
2 # numeric variables
3 library(gapminder)
4 gapminder %>%
5   summarize_if(is.numeric,
6               list(avg = mean))

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

```

1 # get mean and sd for
2 # pop and lifeExp
3
4 gapminder %>%
5   summarize_at(vars(pop, lifeExp),
6               list("avg" = mean,
7                     "std dev" = sd))

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

group_by(): Grouped summaries

group_by() + summarize() |

- 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()**

```

1 # get average life expectancy and gdp by continent
2 gapminder %>%
3   group_by(continent) %>%
4   summarize(avg_life = mean(lifeExp),
5             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

```

1 # track changes in average life expectancy and gdp over time
2 gapminder %>%
3   group_by(year) %>%
4   summarize(mean_life = mean(lifeExp),
5             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)

```

1 # track changes in average life expectancy and gdp over time
2 gapminder %>%
3   group_by(continent, year) %>%
4   summarize(mean_life = mean(lifeExp),
5             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

```

1 gapminder %>%
2   group_by(continent, year) %>%
3   summarize(mean_life = mean(lifeExp),
4             mean_GDP = mean(gdpPercap)) %>%
5   # now pipe this tibble in as data for ggplot!
6   ggplot(data = ., # . pipes the above in (to data layer)
7         aes(x = year,
8               y = mean_life,
9               color = continent)) +
10  geom_path(size = 1) +
11  labs(x = "Year",
12        y = "Average Life Expectancy (Years)",
13        color = "Continent",
14        title = "Average Life Expectancy Over Time") +
15  theme_classic(base_family = "Fira Sans Condensed",
16                base_size = 20)

```

dplyr: Other Useful Commands

tally(): counts for categories

- tally provides counts, best used with group_by for factors

```
1 gapminder %>%
2   tally
# A tibble: 1 × 1
      n
  <int>
1 1704
```

```
1 gapminder %>%
2   group_by(continent) %>%
3   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 filtering by *values*

```
1 gapminder %>%
2   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.
```

```
1 gapminder %>%
2   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`)

```
1 # Get all U.S. life expectancy observations
2 gapminder %>%
3   filter(country == "United States") %>%
4   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
```

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

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

[1] 78.242

1 # 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)

```
1 gapminder %>%
2   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

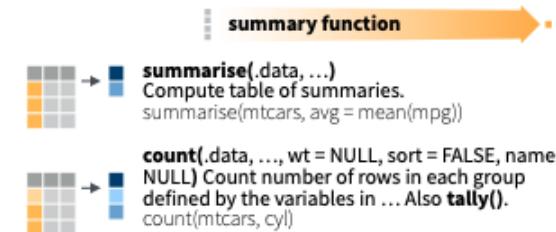


dplyr functions work with pipes and expect tidy data. In tidy data:



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).



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.



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



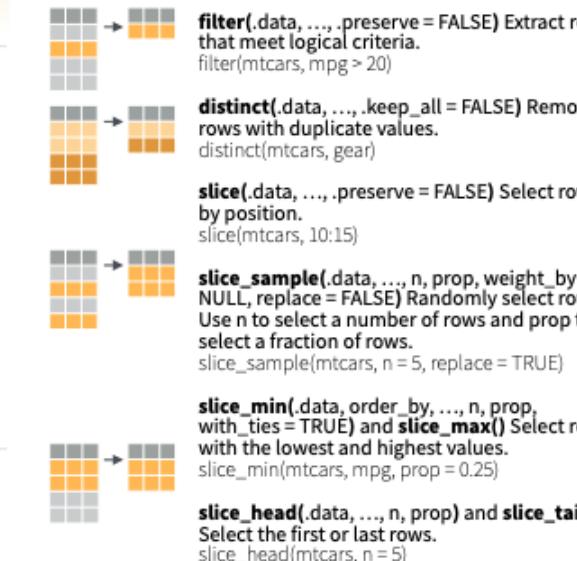
ungroup(x, ...) Returns ungrouped copy of table.
ungroup(mtcars)



Manipulate Cases

EXTRACT CASES

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



Logical and boolean operators to use with filter()

== < <= is.na() %in% | xor()

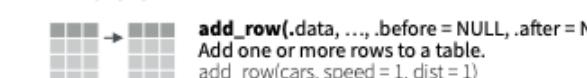
!= > >= is.na() ! &

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

ARRANGE CASES



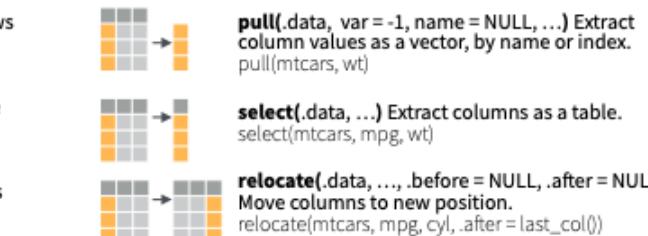
ADD CASES



Manipulate Variables

EXTRACT VARIABLES

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



Use these helpers with select() and across()

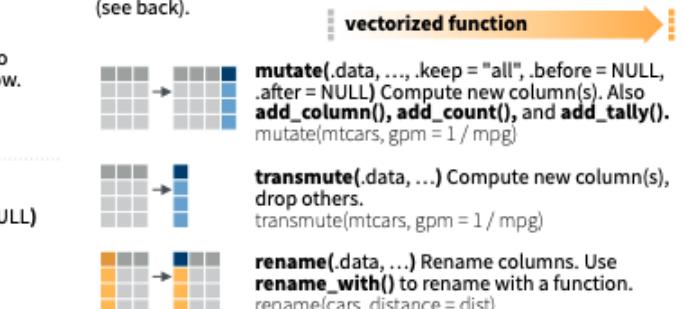
e.g. select(mtcars, mpg:cyl)
contains(match) **num_range**(prefix, range) ;, e.g. mpg:cyl
ends_with(match) **all_of**(x)/**any_of**(x, ..., vars) -, e.g., gear
starts_with(match) **matches**(match) **everything**()

MANIPULATE MULTIPLE VARIABLES AT ONCE



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).



Resources

- *R For Data Science*, Chapter 10: Tibbles
- *R For Data Science*, Chapter 11: Data Import
- R Studio Cheatsheet: Data Import
- *R For Data Science*, Chapter 5: Data Transformation
- R Studio Cheatsheet: Data Wrangling ([New version](#))
- *R For Data Science*, Chapter 12: Tidy Data
- R Studio Cheatsheet: Data Wrangling
- R Studio Cheatsheet: Data Import
- *R For Data Science*, Chapter 13: Relational Data
- R Studio Cheatsheet: Data Transformation

Zahid Asghar

