1.4 — Data Wrangling in the tidyverse

ECON 480 • Econometrics • Fall 2020

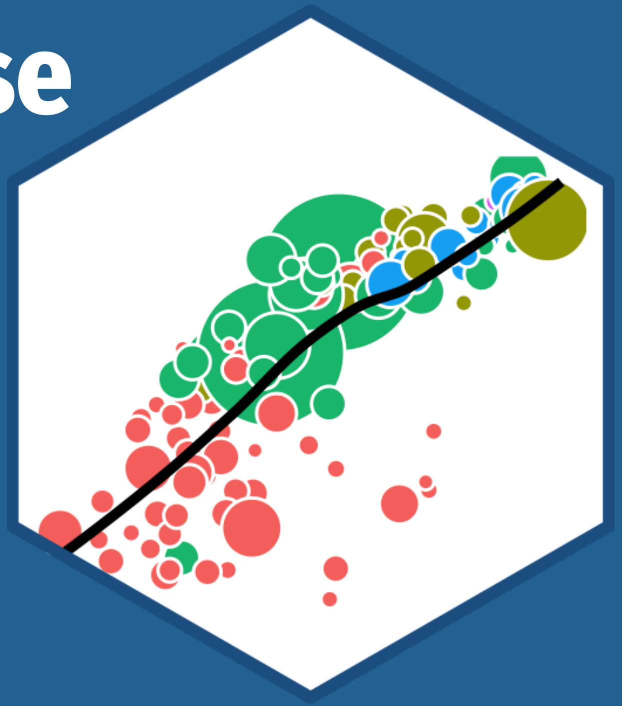
Ryan Safner

Assistant Professor of Economics

safner@hood.edu

ryansafner/metricsF20

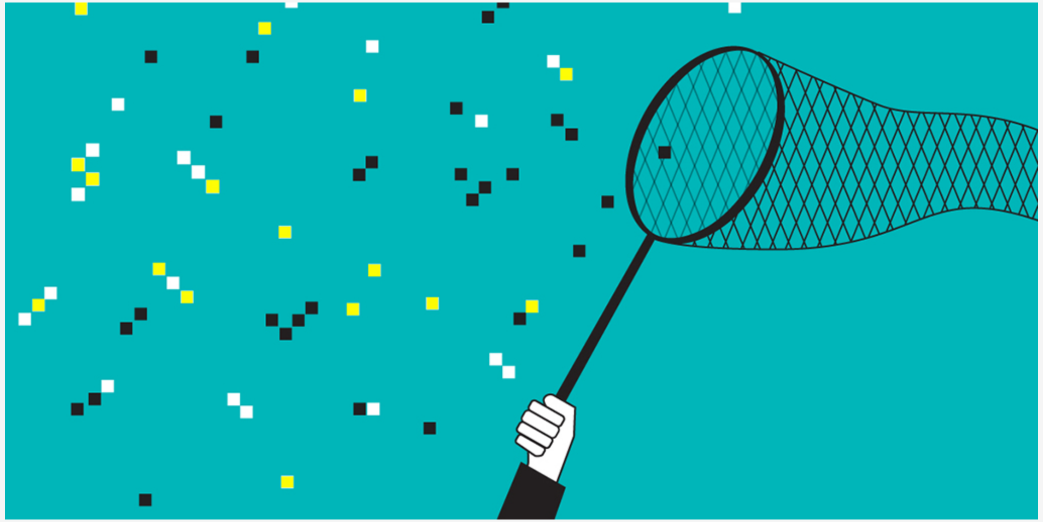
metricsF20.classes.ryansafner.com

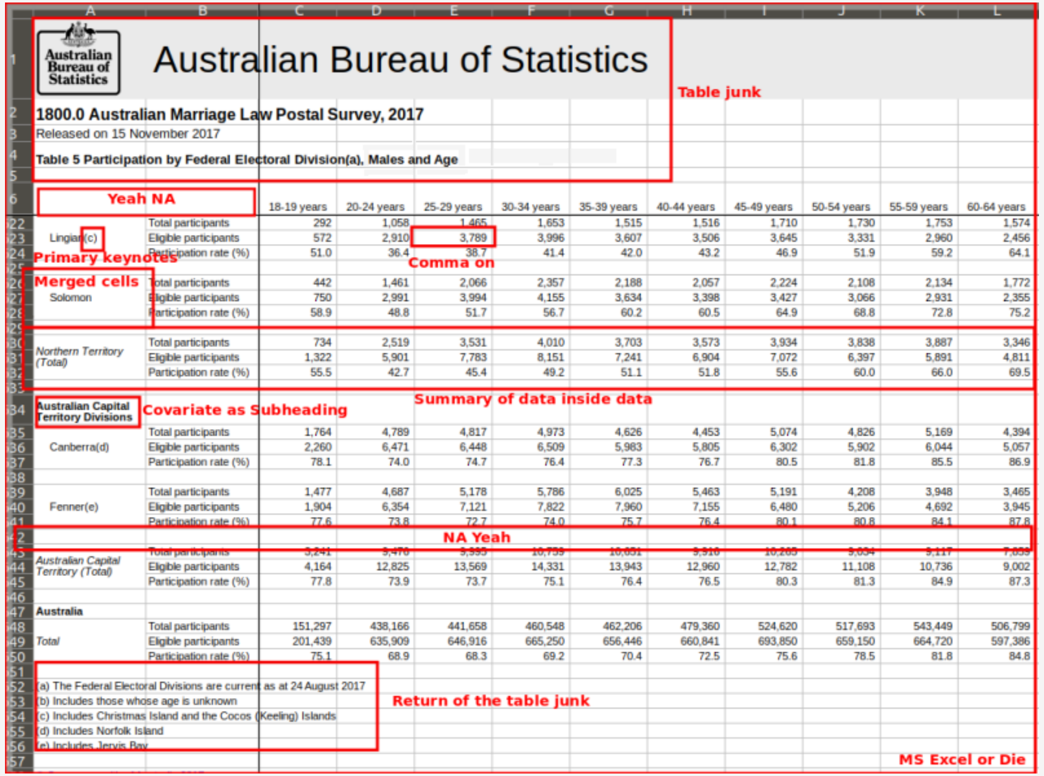


- tibble: friendlier dataframes
- magrittr: piping code
- readr: importing data
- dplyr: wrangling data
- dplyr::filter(): select observations
- dplyr::arrange(): reorder observations
- dplyr::select(): select variables
- dplyr::rename(): rename variables
- dplyr::mutate(): create new variables
- dplyr::summarize(): create statistics
- tidyr: reshaping data
- dplyr: combining datasets

Data Wrangling

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

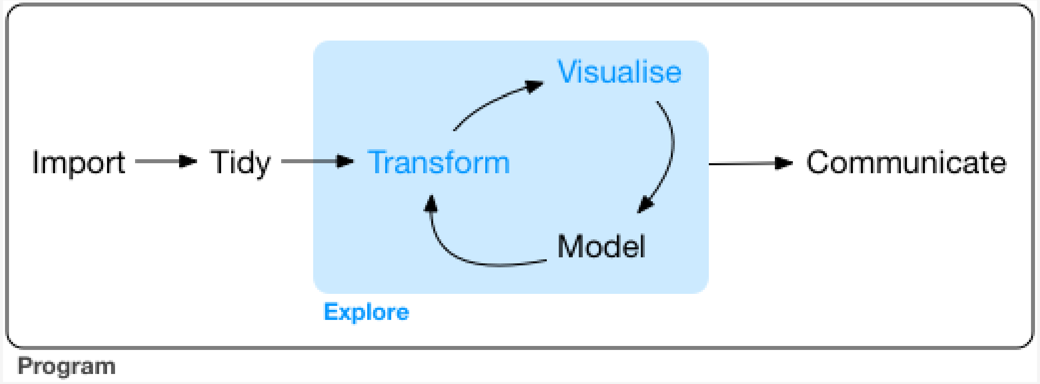




Workflow of a Data Scientist I

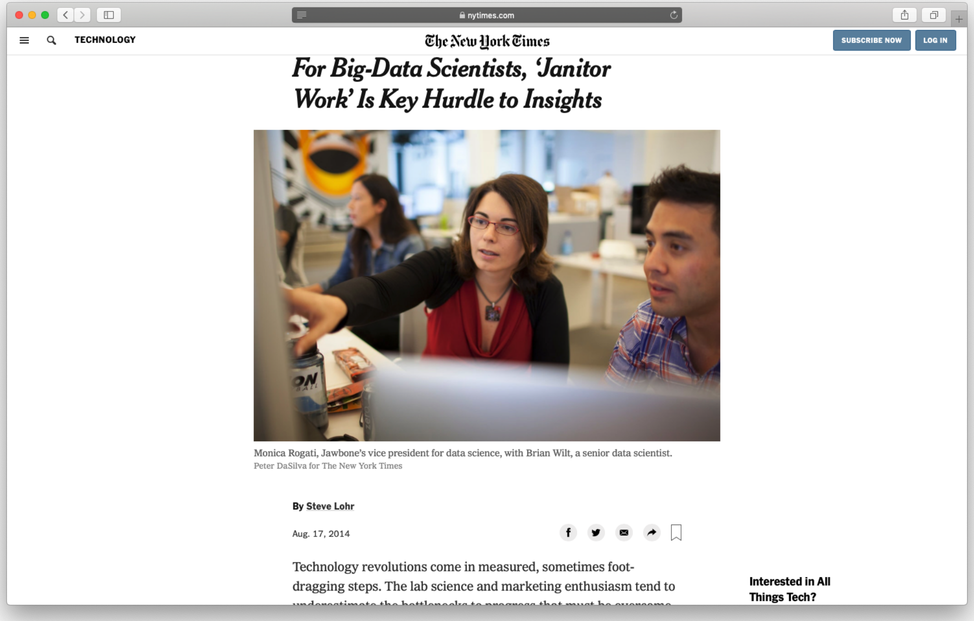
1. Import raw data from out there in the world
2. Tidy it into a form that you can use
3. Explore the data (do these 3 repetitively!)
   - **Transform**
   - **Visualize**
   - **Model**
4. Communicate results to target audience

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


TECHNOLOGY

The New York Times

SUBSCRIBE NOW LOG IN

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

Interested in All Things Tech?

[New York Times]

> "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."

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

- Easiest to just load the core tidyverse all at once
  - First install may take a few minutes - installs a lot of packages!
  - Note loading the tidyverse is "noisy", it will spew a lot of messages
  - Hide them with `suppressPackageStartupMessages()` and insert `library()` command inside

```r
# install for first time
# install.packages("tidyverse") # this takes a few minutes and may give several prompts

# load tidyverse
suppressPackageStartupMessages(library("tidyverse"))
```

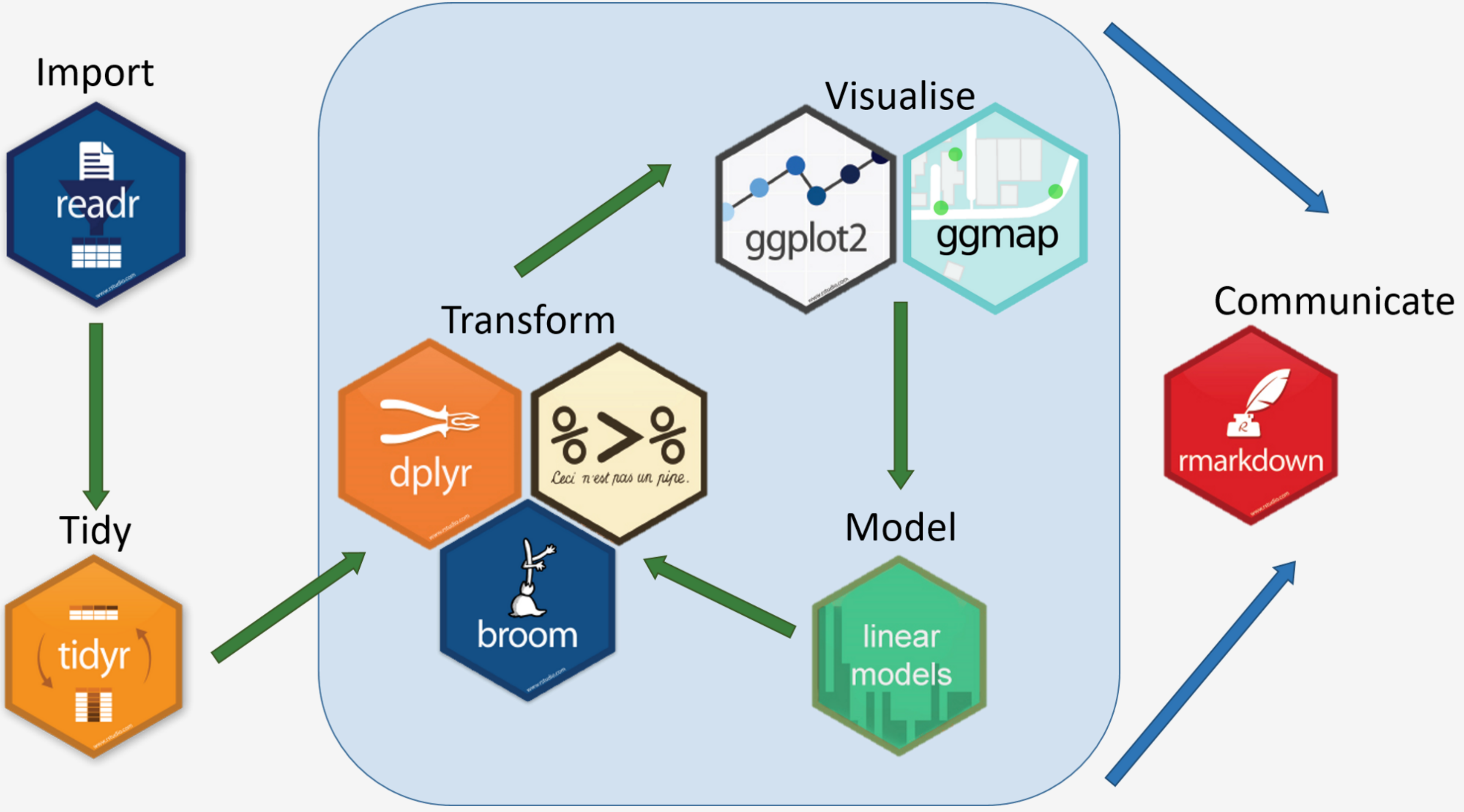
The tidyverse III

- `tidyverse` contains a lot of packages, not all are loaded automatically

```r
tidyverse_packages()
```

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
##  [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "tidyverse"
```

Your Workflow in the tidyverse:



Tidyverse Packages

- We will make **extensive** use of (and talk today about):

1. `tibble` for friendlier dataframes
2. `magrittr` for "pipeable" code
3. `readr` for importing data
4. `dplyr` for data wrangling
5. `tidyr` for tidying data
6. `ggplot2` for plotting data (we've already covered)

- We will (or might) later look at:

1. `broom` for tidy regression (not part of core tidyverse)
2. `forcats` for working with factors
3. `stringr` for working with strings
4. `lubridate` for working with dates and times
5. `purrr` for iteration

tibble: friendlier dataframes

tibble I

- `tibble` converts all `data.frames` into a *friendlier* version called `tibbles` (or `tbl_df`)

tibble II

```
diamonds
```

```
## # A tibble: 53,940 x 7
##    carat cut       color clarity depth table price
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>
##  1 0.23  Ideal     E     SI2      61.5    55   326
##  2 0.21  Premium   E     SI1      59.8    61   326
##  3 0.23  Good      E     VS1      56.9    65   327
##  4 0.290 Premium   I     VS2      62.4    58   334
##  5 0.31  Good      J     SI2      63.3    58   335
##  6 0.24  Very Good J     VVS2     62.8    57   336
##  7 0.24  Very Good I     VVS1     62.3    57   336
##  8 0.26  Very Good H     SI1      61.9    55   337
##  9 0.22  Fair      E     VS2      65.1    61   337
## 10 0.23  Very Good H     VS1      59.4    61   338
## # … with 53,930 more rows
```

- Prints much nicer output
- Shows a bit of the `str`ucture:
    - `nrow() x ncol()`
    - `<dbl>` is numeric ("double")
    - `<ord>` is an ordered factor
    - `<int>` is an integer
- Fundamental grammar of tidyverse:
    1. start with a tibble
    2. run a function on it
    3. output a new tibble

tibble III

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

```r
as_tibble(mpg) # take built-in dataframe mpg
```

- 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"))

example
```

```
## # A tibble: 3 x 3
##       x      y colors
##   <dbl>  <dbl> <chr>
## 1     2  1.40  orange
## 2     4 -0.439 green
## 3     6 -0.896 blue
```

tibble IV

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

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

example_2
```

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

magrittr: piping code

magrittr I

- The `magrittr` package allows us to use the **"pipe" operator** (`%>%`)†

- `%>%` "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"

† Keyboard shortcuts in R Studio: `CTRL+Shift+M` (Windows) or `Cmd+Shift+M` (Mac)

magrittr II

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

$$g(f(x))$$

  - i.e. take `x` and perform function `f()` on `x` and then take that result and perform function `g()` on it

- With pipes, read operations from left → right:

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

take `x` and then perform function `f` on it, then perform function `g` on that result

- Read `%>%` mentally as "and then"

magrittr III

Example

$$ln(exp(x))$$

- First, exponentiate x, then take the natural log of that (resulting in just x)
- In pipes:

```
x %>% exp() %>% ln()
```

magrittr IV

Ceci n'est pas un pipe

Example

- Sequence: find keys, unlock car, drive to school, park
- Using nested functions in pseudo-"code":

```
park(drive(start_car(find("keys")), to = "campus"))
```

- Using pipes:

```
find("keys") %>%
  start_car() %>%
  drive(to = "campus") %>%
  park()
```

magrittr: Simple Example

```r
# look at top 6 rows
head(gapminder)

# use pipe instead
gapminder %>% head()
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdp
##   <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
```

magrittr: More Involved Example

- These two methods produce the same output (average hightway mpg of Audi cars)

- Without the pipe

```r
summarise(group_by(filter(mpg, manufacturer=="audi"), model), hwy_mean = mean(hwy))
```

- Using the pipe

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

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

readr: importing data

readr

- `readr` helps load common spreadsheet files (`.csv`, `.tsv`) with simple commands:
- `read_*(path/to/my_data.*)`
  - where `*` can be `.csv` or `.tsv`
- Often this is enough, but many more customizations possible
- You 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

Readxl and Haven: When Readr isn't Enough

- 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)

Common Import Issues I

- Most common: *"where the hell is my data file"??*

- Recall `R` looks for files to `read_*()` in the default working directory (check what it is with `getwd()`, change it with `setwd()`)

- 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")
```

Common Import Issues II

- Most common: *"where the hell is my data file"??*

- Recall `R` looks for files to `read_*()` in the default working directory (check what it is with `getwd()`, change it with `setwd()`)

- 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"

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

```
# 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 III

- **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!

dplyr: wrangling data

dplyr I

- `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)`[†]
- Three 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

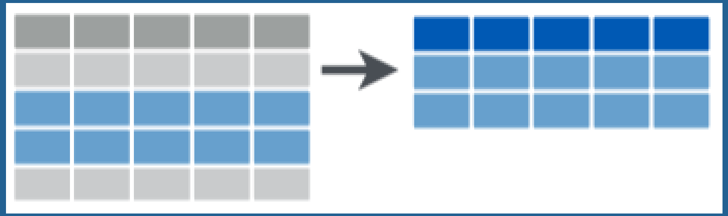
[†] With the pipe, even simpler: `df %>% verb(conditions)`

dplyr II

- 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

dplyr::filter(): select observations



dplyr::filter()

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

```r
# look only at African observations
# syntax without the pipe
filter(gapminder, continent=="Africa")
```

```r
# using the pipe

gapminder %>%
  filter(continent == "Africa")
```

```
## # A tibble: 624 x 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
```

dplyr: saving and storing outputs 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

```r
# base syntax
africa <- filter(gapminder,
                 continent=="Africa")
```

```r
# using the pipe
africa <- gapminder %>%
  filter(continent == "Africa")
```

```r
# look at new tibble
africa
```

```
## # A tibble: 624 x 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
```

dplyr: saving and storing outputs II

- If you want to *both* store and view the output at the same time, wrap the command in parentheses!

```r
(africa <- gapminder %>%
  filter(continent == "Africa"))
```

```
## # A tibble: 624 x 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.
```

dplyr: saving and storing outputs III

- If you were to assign the output to the original `tibble`, it would *overwrite* the original!

```
# base syntax
gapminder <- filter(gapminder,
                    continent=="Africa")
```

```
# using the pipe
gapminder <- gapminder %>%
  filter(continent == "Africa")

# this overwrites gapminder!
```

dplyr Conditionals

- 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`[†]
    - `>`, `<`: greater than, less than
    - `>=`, `<=`: greater than or equal to, less than or equal to
    - `==`[‡], `!=`: is equal to[‡], is not equal to
    - `%in%`: is a member of some defined set ($\in$)
    - `&`: AND (commas also work instead)
    - `|`: OR
    - `!`: not

[†] See `?Comparison` and `?Base::Logic`.

[‡] Recall one `=` *assigns* values to an object, two `==` *tests* an object for a condition!

dplyr::filter() with Conditionals

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

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

dplyr::filter() with Conditionals II

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

```
## # A tibble: 714 x 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
```

dplyr::filter() with Conditionals III

```r
# 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 x 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.
```

dplyr::arrange(): reorder observations

dplyr::arrange() I

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

```
# order by smallest to largest pop
# syntax without the pipe
arrange(gapminder, pop)
```

```
# using the pipe

gapminder %>%
  arrange(pop)
```

```
## # A tibble: 1,704 x 6
##    country               continent  year lifeExp   pop gdpPerca
##    <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
```

dplyr::arrange() II

- 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
# syntax without the pipe
arrange(gapminder, year, pop)
```

```r
# using the pipe

gapminder %>%
  arrange(year, pop)
```

```
## # A tibble: 1,704 x 6
##    country               continent  year lifeExp    pop gdpPerc
##    <fct>                 <fct>     <int>   <dbl>  <int>     <db
##  1 Sao Tome and Principe Africa     1952    46.5  60011      88
##  2 Djibouti              Africa     1952    34.8  63149     267
##  3 Bahrain               Asia       1952    50.9 120447     986
##  4 Iceland               Europe     1952    72.5 147962     726
##  5 Comoros               Africa     1952    40.7 153936     110
##  6 Kuwait                Asia       1952    55.6 160000   10838
##  7 Equatorial Guinea     Africa     1952    34.5 216964      37
##  8 Reunion               Africa     1952    52.7 257700     271
##  9 Gambia                Africa     1952    30   284320      48
## 10 Swaziland             Africa     1952    41.4 290243     114
## # … with 1,694 more rows
```

dplyr::arrange() III

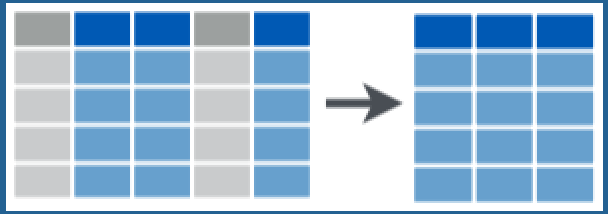
- Use `desc()` to re-order in the opposite direction

```r
# order by largest to smallest pop
# syntax without the pipe
arrange(gapminder, desc(pop))
```

```r
# using the pipe

gapminder %>%
  arrange(desc(pop))
```

```
## # A tibble: 1,704 x 6
##    country continent  year lifeExp        pop gdpPercap
##    <fct>   <fct>     <int>   <dbl>      <int>     <dbl>
##  1 China   Asia       2007    73.0 1318683096     4959.
##  2 China   Asia       2002    72.0 1280400000     3119.
##  3 China   Asia       1997    70.4 1230075000     2289.
##  4 China   Asia       1992    68.7 1164970000     1656.
##  5 India   Asia       2007    64.7 1110396331     2452.
##  6 China   Asia       1987    67.3 1084035000     1379.
##  7 India   Asia       2002    62.9 1034172547     1747.
##  8 China   Asia       1982    65.5 1000281000      962.
##  9 India   Asia       1997    61.8  959000000     1459.
## 10 China   Asia       1977    64.0  943455000      741.
## # … with 1,694 more rows
```

dplyr::select(): select variables



dplyr::select() I

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

```r
# keep only country, year,
# and population variables
# syntax without the pipe
select(gapminder, country, year, pop)
```

```r
# using the pipe

gapminder %>%
  select(country, year, pop)
```

```
## # A tibble: 1,704 x 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
```

dplyr::select() II

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

```r
# keep all *except* gdpPercap
# syntax without the pipe
select(gapminder, -gdpPercap)
```

```r
# using the pipe

gapminder %>%
  select(-gdpPercap)
```

```
## # A tibble: 1,704 x 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
```

dplyr::select() III

- `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
# keep all and move pop first
# syntax without the pipe
select(gapminder, pop, everything())
```

```r
# using the pipe

gapminder %>%
  select(pop, everything())
```

```
## # A tibble: 1,704 x 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
```

dplyr::select() IV

- `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 x 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
```

```r
# keep country and all variables
# containing "per"

gapminder %>%
  select(country, contains("per"))
```

```
## # A tibble: 1,704 x 2
##    country     gdpPercap
##    <fct>           <dbl>
##  1 Afghanistan      779.
##  2 Afghanistan      821.
##  3 Afghanistan      853.
##  4 Afghanistan      836.
##  5 Afghanistan      740.
##  6 Afghanistan      786.
```

dplyr::rename(): rename variables

dplyr::rename()

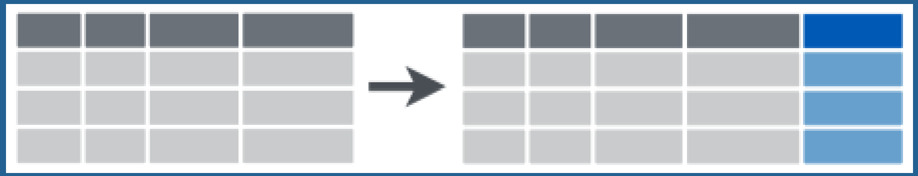
- `rename` changes the name of a variable (column)
  - Format: `new_name = old_name`

```r
# rename gdpPercap to GDP
# syntax without the pipe
rename(gapminder, GDP = gdpPercap)
```

```r
# using the pipe

gapminder %>%
  rename(GDP = gdpPercap)
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      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
```

dplyr::mutate(): create new variables



dplyr::mutate()

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

dplyr::mutate() II

- Three major types of mutates:

1. Create a variable that is a specific value (often categorical)

```r
# create variable "europe" if country
# is in Europe
# syntax without the pipe
mutate(gapminder,
       europe = ifelse(continent == "Europe",
                       yes = "In Europe",
                       no = "Not in Europe"))
```

```r
# using the pipe

gapminder %>%
  mutate(europe = ifelse(continent == "Europe",
                         yes = "In Europe",
                         no = "Not in Europe"))
```

```
## # A tibble: 1,704 x 4
##    country     continent  year europe
##    <fct>       <fct>     <int> <chr>
##  1 Afghanistan Asia       1952 Not in Europe
##  2 Afghanistan Asia       1957 Not in Europe
##  3 Afghanistan Asia       1962 Not in Europe
##  4 Afghanistan Asia       1967 Not in Europe
##  5 Afghanistan Asia       1972 Not in Europe
##  6 Afghanistan Asia       1977 Not in Europe
##  7 Afghanistan Asia       1982 Not in Europe
##  8 Afghanistan Asia       1987 Not in Europe
##  9 Afghanistan Asia       1992 Not in Europe
## 10 Afghanistan Asia       1997 Not in Europe
## # … with 1,694 more rows
```

dplyr::mutate() III

- Three major types of mutates:

1. Create a variable that is a specific value (often categorical)
2. Change an existing variable (often rescaling)

```
# create population in millions
# syntax without the pipe
mutate(gapminder,
       pop_mil = pop / 1000000)
```

```
# using the pipe

gapminder %>%
  rename(pop_mil = pop / 1000000)
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop pop_mil
##    <fct>       <fct>     <int>   <dbl>    <int>   <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333    8.43
##  2 Afghanistan Asia       1957    30.3  9240934    9.24
##  3 Afghanistan Asia       1962    32.0 10267083   10.3
##  4 Afghanistan Asia       1967    34.0 11537966   11.5
##  5 Afghanistan Asia       1972    36.1 13079460   13.1
##  6 Afghanistan Asia       1977    38.4 14880372   14.9
##  7 Afghanistan Asia       1982    39.9 12881816   12.9
##  8 Afghanistan Asia       1987    40.8 13867957   13.9
##  9 Afghanistan Asia       1992    41.7 16317921   16.3
## 10 Afghanistan Asia       1997    41.8 22227415   22.2
```

dplyr::mutate() IV

- 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

```r
# create GDP variable from gdpPercap
# and pop, in billions
# syntax without the pipe
mutate(gapminder,
       GDP = ((gdpPercap * pop)/100000
```

```r
# using the pipe

gapminder %>%
  mutate(GDP = ((gdpPercap * pop)/1000
```

```
## # A tibble: 1,704 x 6
##    country     continent  year      pop gdpPercap   GDP
##    <fct>       <fct>     <int>    <int>     <dbl> <dbl>
##  1 Afghanistan Asia       1952  8425333      779.  6.57
##  2 Afghanistan Asia       1957  9240934      821.  7.59
##  3 Afghanistan Asia       1962 10267083      853.  8.76
##  4 Afghanistan Asia       1967 11537966      836.  9.65
##  5 Afghanistan Asia       1972 13079460      740.  9.68
##  6 Afghanistan Asia       1977 14880372      786. 11.7
##  7 Afghanistan Asia       1982 12881816      978. 12.6
##  8 Afghanistan Asia       1987 13867957      852. 11.8
```

dplyr::mutate() V

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

```
gapminder %>% head(., 2)
```

```
## # A tibble: 2 x 6
##   country     continent  year lifeExp     pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8 8425333      779.
## 2 Afghanistan Asia       1957    30.3 9240934      821.
```

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

```
## # A tibble: 1,704 x 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.
```

dplyr::mutate(): Multiple Variables

- Can create multiple new variables with commas:

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

```
## # A tibble: 1,704 x 8
##    country     continent  year lifeExp      pop gdpPercap        GDP pop_millions
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>      <dbl>        <dbl>
##  1 Afghanist… Asia       1952    28.8  8425333      779.    6.57e 9         8.43
##  2 Afghanist… Asia       1957    30.3  9240934      821.    7.59e 9         9.24
##  3 Afghanist… Asia       1962    32.0 10267083      853.    8.76e 9        10.3
##  4 Afghanist… Asia       1967    34.0 11537966      836.    9.65e 9        11.5
##  5 Afghanist… Asia       1972    36.1 13079460      740.    9.68e 9        13.1
##  6 Afghanist… Asia       1977    38.4 14880372      786.    1.17e10        14.9
##  7 Afghanist… Asia       1982    39.9 12881816      978.    1.26e10        12.9
##  8 Afghanist… Asia       1987    40.8 13867957      852.    1.18e10        13.9
##  9 Afghanist… Asia       1992    41.7 16317921      649.    1.06e10        16.3
## 10 Afghanist… Asia       1997    41.8 22227415      635.    1.41e10        22.2
## # … with 1,694 more rows
```

dplyr::transmute()

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

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

```
## # A tibble: 1,704 x 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
```

dplyr::mutate(): Conditionals

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

```
gapminder %>%
  select(country, year, lifeExp) %>%
  mutate(long_1 = lifeExp > 70,
         long_2 = ifelse(lifeExp > 70, "Long", "Short"))
```

```
## # A tibble: 1,704 x 5
##    country      year lifeExp long_1 long_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
```

dplyr::mutate(): 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 x 5
##    country      year lifeExp dog_years comment
##    <fct>       <int>   <dbl>     <dbl> <chr>
##  1 Afghanist…  1952    28.8      202. Life expectancy in Afghanistan is 201.607…
##  2 Afghanist…  1957    30.3      212. Life expectancy in Afghanistan is 212.324…
##  3 Afghanist…  1962    32.0      224. Life expectancy in Afghanistan is 223.979…
##  4 Afghanist…  1967    34.0      238. Life expectancy in Afghanistan is 238.14 …
##  5 Afghanist…  1972    36.1      253. Life expectancy in Afghanistan is 252.616…
##  6 Afghanist…  1977    38.4      269. Life expectancy in Afghanistan is 269.066…
##  7 Afghanist…  1982    39.9      279. Life expectancy in Afghanistan is 278.978…
##  8 Afghanist…  1987    40.8      286. Life expectancy in Afghanistan is 285.754…
```

dplyr::mutate(): case_when()

- `case_when` creates a new variable with values that are conditional on values of other variables (e.g., "if/else")
  - Last argument: `TRUE`: when

```
gapminder %>%
  mutate(European = case_when(
    continent == "Europe" ~ "Aye",
    TRUE ~ "Nay"
  ))
```

```
## # A tibble: 1,704 x 7
##    country     continent  year lifeExp      pop gdpPercap European
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl> <chr>
##  1 Afghanistan Asia       1952    28.8  8425333      779. Nay
##  2 Afghanistan Asia       1957    30.3  9240934      821. Nay
##  3 Afghanistan Asia       1962    32.0 10267083      853. Nay
##  4 Afghanistan Asia       1967    34.0 11537966      836. Nay
##  5 Afghanistan Asia       1972    36.1 13079460      740. Nay
```

dplyr::mutate(): scoped 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 x 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.
```

dplyr::mutate(): scoped 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

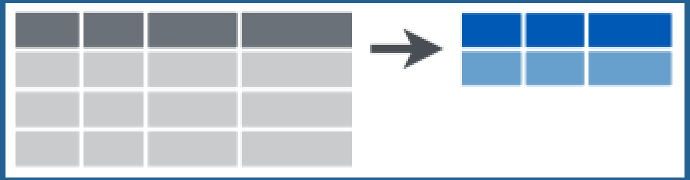
```
# make all factor variables uppercase
gapminder %>%
  mutate_if(is.factor, toupper)
```

```
## # A tibble: 1,704 x 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.
```

dplyr::mutate()

- Don't forget to assign the output to a new `tibble` (or overwrite original) if you want to "save" the new variables!

dplyr::summarize(): create statistics



dplyr::summarize() I

- `summarize`[†] 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
# call it avg_LE
summarize(gapminder,
          avg_LE = mean(lifeExp))
```

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

```
# using the pipe

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

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

dplyr::summarize() II

- 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. These commands require nothing to be in parentheses!

dplyr::summarize() II

- Useful `summarize()` commands (continued):

Command	Does
`min()`	Minimum value of a variable
`max()`	Maximum value of a variable
`quantile(., 0.25)`[+]	Specified percentile (example `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 `.` is where you would put your variable name.

dplyr::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 x 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 x 2
##     year     n
##    <int> <int>
##  1  1952   142
```

dplyr::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 x 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 x 1
##   Over_70
##     <dbl>
## 1   0.585
```

dplyr::summarize() Multiple Variables

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

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

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

```r
# using the pipe

gapminder %>%
  summarize(avg_LE = mean(lifeExp),
            avg_GDP = mean(gdpPercap))
```

dplyr::summarize() Multiple Statistics

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

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

```
## # A tibble: 1 x 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
```

```r
# using the pipe

gapminder %>%
  summarize(obs = n(),
          avg_LE = mean(lifeExp),
          sd_LE = sd(lifeExp),
          min_LE = min(lifeExp),
```

dplyr::summarize() Multiple Statistics

- "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 x 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 x 4
##     pop_avg lifeExp_avg `pop_std dev` `lifeExp_std
##       <dbl>       <dbl>         <dbl>
## 1 29601212.        59.5    106157897.
```

dplyr::summarize() with group_by() 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(mean_life = mean(lifeExp),
            mean_GDP = mean(gdpPercap))
```

```
## # A tibble: 5 x 3
##   continent mean_life mean_GDP
##   <fct>         <dbl>    <dbl>
## 1 Africa         48.9    2194.
```

dplyr::summarize() with group_by() 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 x 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.
```

dplyr::summarize() with group_by() III

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

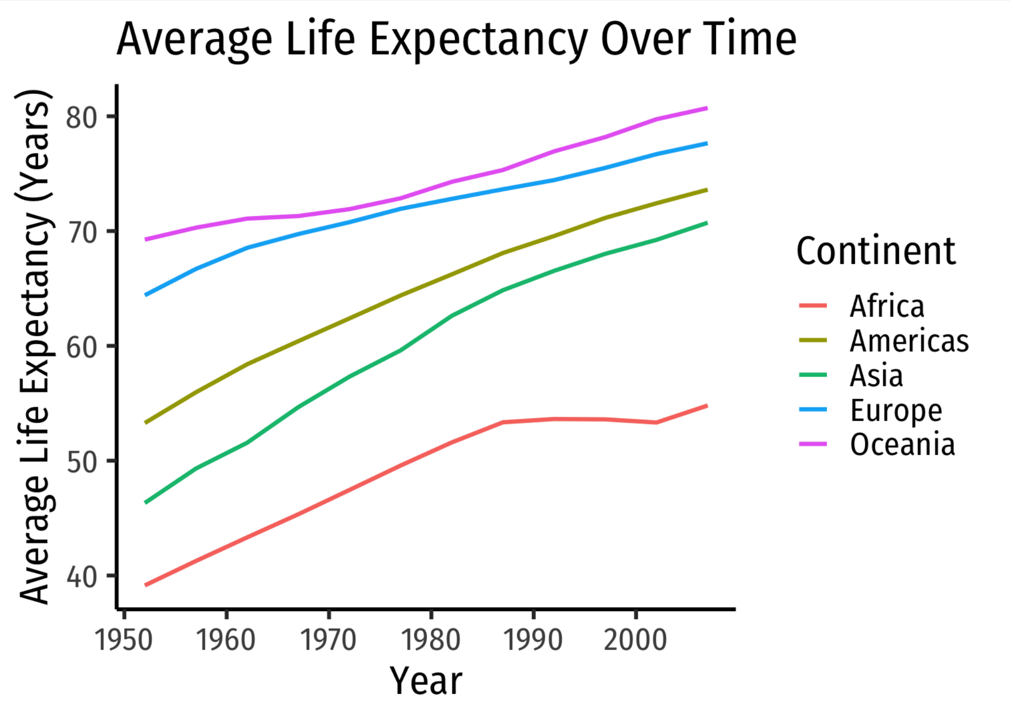
```
# track changes in average life expectancy and gdp by continent over time

gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_life = mean(lifeExp),
            mean_GDP = mean(gdpPercap))
```

```
## # A tibble: 60 x 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.
```

Example: Piping Across Packages

- `tidyverse` uses same grammar and design philosophy
- **Example**: graphing change in average life expectancy by continent over time

```
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 = ., # . stands in for stuff ^!
     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 Conden
```



dplyr: Other Useful Commands I

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

```
gapminder %>%
  tally
```

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

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

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

dplyr: Other Useful Commands II

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

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

```
## # A tibble: 3 x 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.
```

dplyr: Other Useful Commands III

- `pull()` extracts a column from a `tibble` (just like `$`)

```
# 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
```

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

```
## [1] 78.242
```

dplyr: Other Useful Commands IV

- `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 x 1
##    country
##    <fct>
##  1 Afghanistan
##  2 Albania
##  3 Algeria
##  4 Angola
##  5 Argentina
##  6 Australia
##  7 Austria
##  8 Bahrain
##  9 Bangladesh
## 10 Belgium
```

tidyr: reshaping data

tidyr: reshaping and tidying data

- `tidyr` helps reshape data into more usable format
- **"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**‡

- Spend less time fighting your tools and more time on analysis!

† This is the namesake of the `tidyverse`: all associated packages and functions use or require this data format!

‡ Alternatively, sometimes rule 3 is "every value is its own cell."

tidyr: Tidy Data

- "tidy" data $\neq$ 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

tidyr::gather() wide to long I

```r
# make example untidy data
ex_wide<-tribble(
  ~"Country", ~"2000", ~"2010",
  "United States", 140, 180,
  "Canada", 102, 98,
  "China", 111, 123
)
ex_wide
```

```
## # A tibble: 3 x 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!
  - Each row represents *two* observations (one in 2000 and one in 2010)!

tidyr::gather() wide to long II

```r
# make example untidy data
ex_wide<-tribble(
  ~"Country", ~"2000", ~"2010",
  "United States", 140, 180,
  "Canada", 102, 98,
  "China", 111, 123
)
ex_wide
```

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

- We need to `gather()` these columns into a new pair of variables
  - set of columns that represent values, not variables (`2000` and `2010`)
  - `key`: name of variable whose values form the column names (we'll call it the `year`)
  - `value`: name of the variable whose values are spread over the cells (we'll call it number of `cases`)

tidyr::gather() wide to long III

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

```r
ex_wide
```

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

```r
ex_wide %>% gather("2000","2010",
              key = "year",
              value = "cases")
```

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

tidyr::spread() long to wide I

```r
ex_long # example I made (code hidden)
```

```
## # A tibble: 12 x 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
## 12 China          2010 Population  1241
```

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

tidyr::spread() long to wide II

```
ex_long # example I made (code hidden)
```

```
## # A tibble: 12 x 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
## 12 China          2010 Population  1241
```

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

tidyr::spread() long to wide III

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

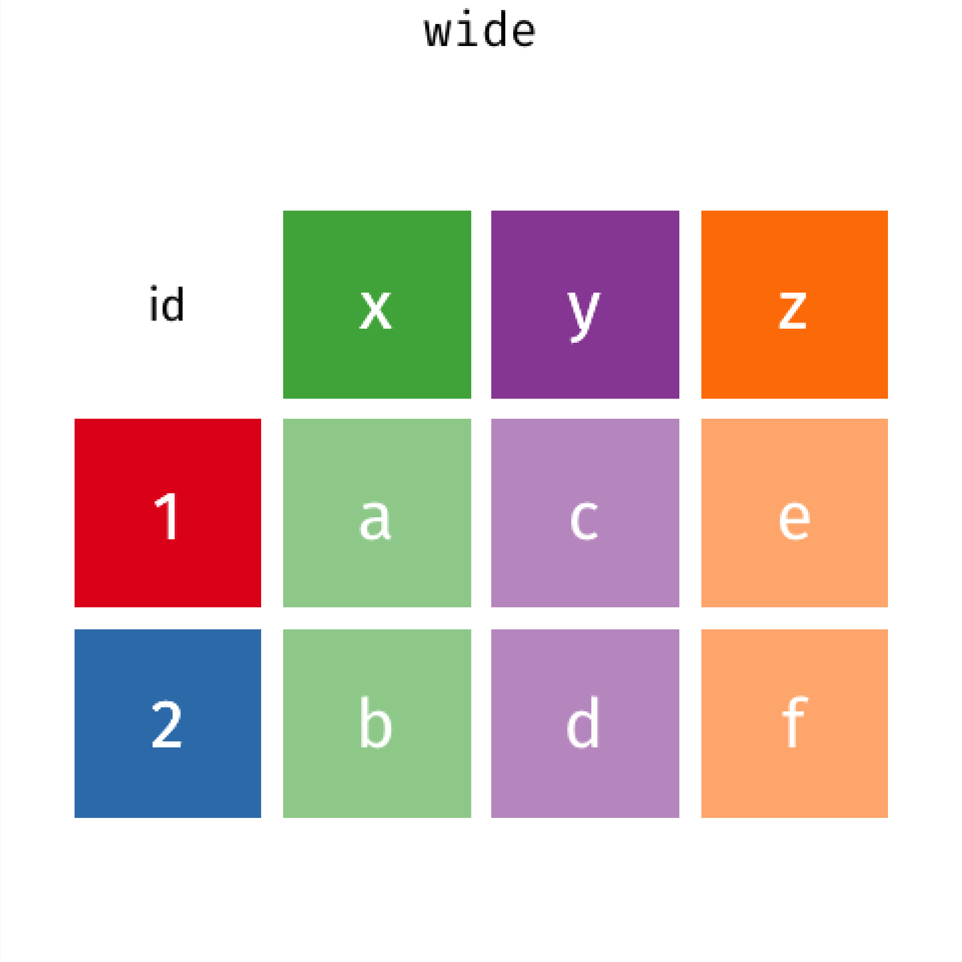
```r
ex_long
```

```
## # A tibble: 12 x 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 %>% spread(key = "Type",
                   value = "Count")
```

```
## # A tibble: 6 x 4
##   Country        Year Cases Population
##   <chr>         <dbl> <dbl>      <dbl>
## 1 Canada         2000   102        110
## 2 Canada         2010    98        121
## 3 China          2000   111       1201
## 4 China          2010   123       1241
## 5 United States  2000   140        300
## 6 United States  2010   180        310
```

tidyr



* Image from Garrick Aden-Buie's excellent [tidyexplain]

Combining Datasets

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA
d	w	NA	1

Combining Datasets

- Often, data doesn't come from just one source, but several sources

- We can combine datasets into a single dataframe (tibble) using `dplyr` commands in several ways:

  1. `bind` dataframes together by row or by column
     - `bind_rows()` adds observations (rows) to existing dataset[1]
     - `bind_cols()` adds variables (columns) to existing dataset[2]
  2. `join` two dataframes by designating variable(s) as `key` to match rows by identical values of that `key`

† Note the columns must be identical between the original dataset and the new observations

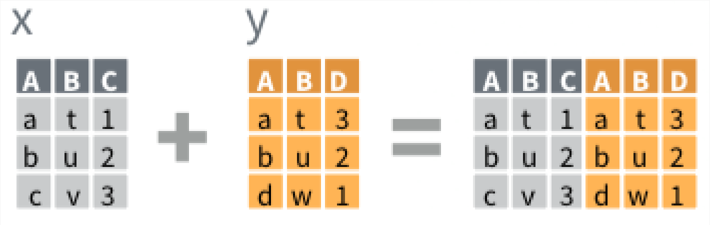
‡ Note the rows must be identical between original dataset and new variable

Two *Similar* Datasets I

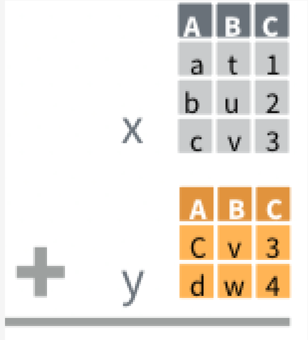
- Sometimes you want to add rows (observations) or columns (variables) that happen to match up perfectly
  - New observations contain all the same variables as existing data
  - OR
  - New variables contain all the same observations as existing data
- In this case, simply using `bind_*(old_df, new_df)` will work
  - `bind_columns(old_df, new_df)` adds columns from `new_df` to `old_df`
  - `bind_rows(old_df, new_df)` adds rows from `new_df` to `old_df`

Two *Similar* Datasets II

`bind_columns()` (Variables)

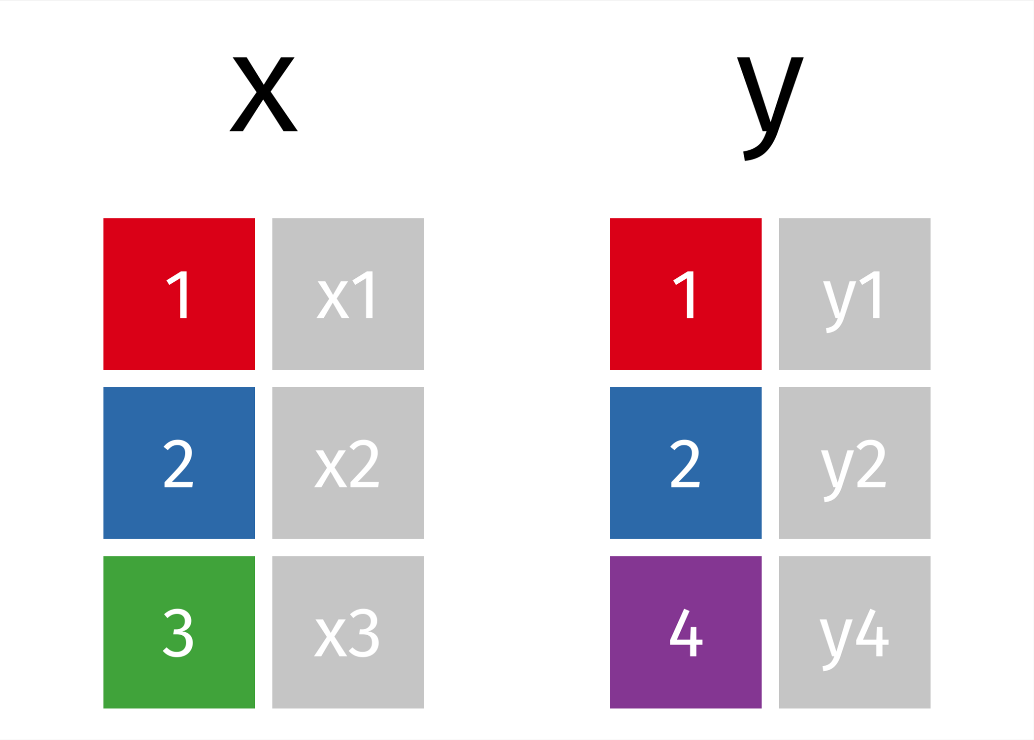


`bind_rows()` (Observations)



Two *Different* Datasets

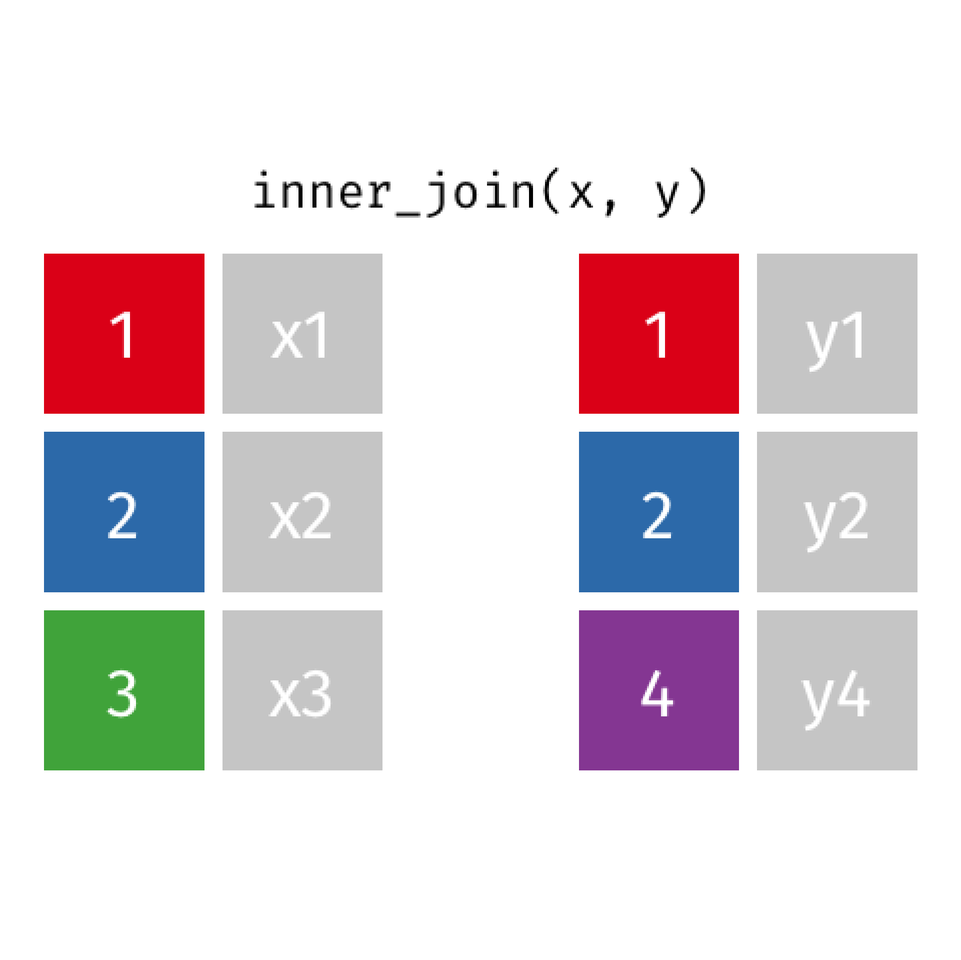
- For the following examples, consider the following two dataframes, `x` and `y` *
  - each has one unique variable, `x$x` and `y$y`
  - both have values for observations `1` and `2`
  - `x` has observation `3` which `y` does not have
  - `y` has observation `4` which `x` does not have
- We next consider the ways we can merge dataframes `x` and `y` into a single dataframe



* Images on all following slides come from Garrick Aden-Buie's excellent [tidyexplain]

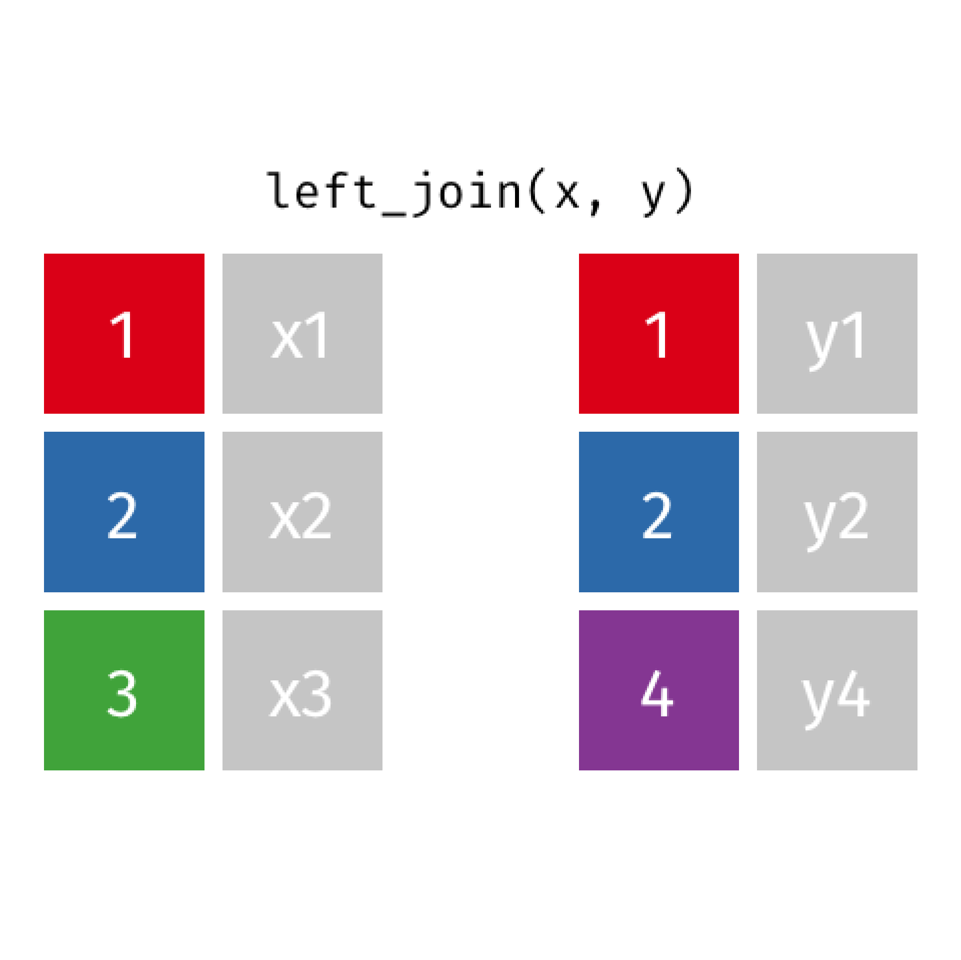
Inner-Join

- Merge columns from `x` and `y` for which there are matching rows
  - Rows in `x` with no match in `y` (3) will be dropped
  - Rows in `y` with no match in `x` (4) will be dropped



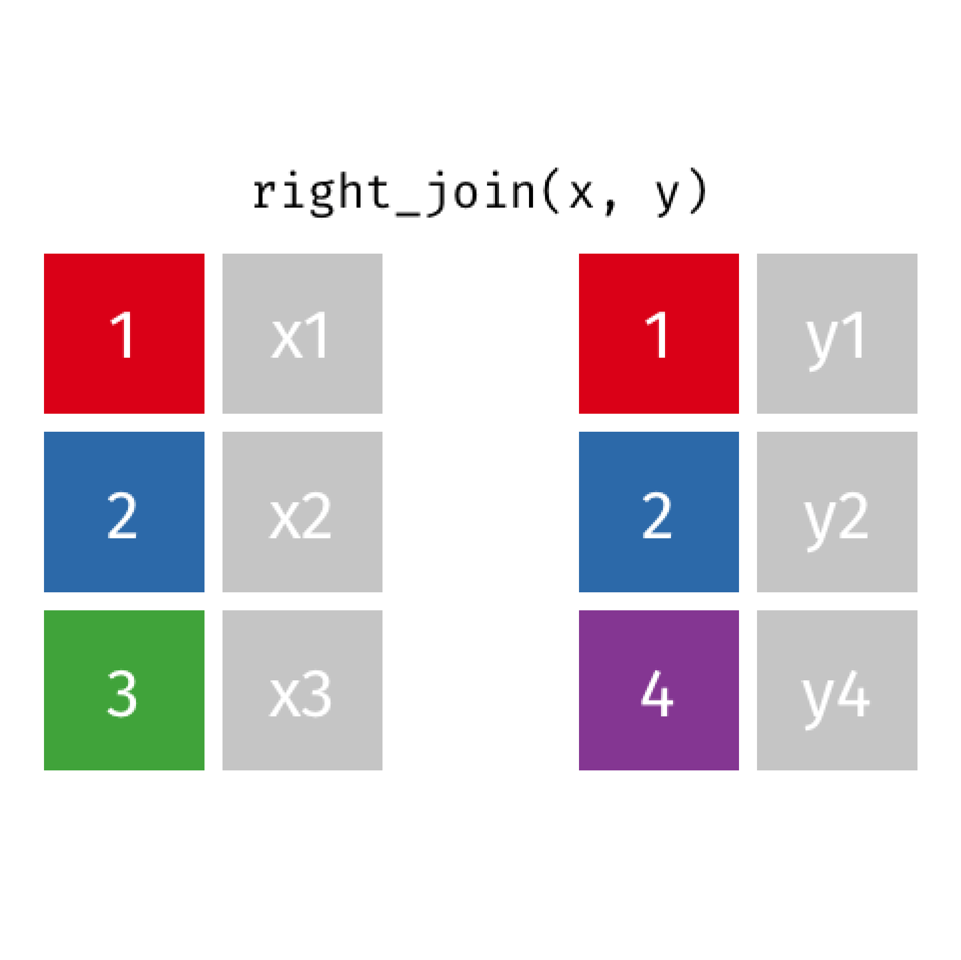
Left-Join

- Start with all rows from `x` and add all columns from `y`
    - Rows in `x` with no match in `y` (3) will have `NA`s
    - Rows in `y` with no match in `x` (4) will be dropped



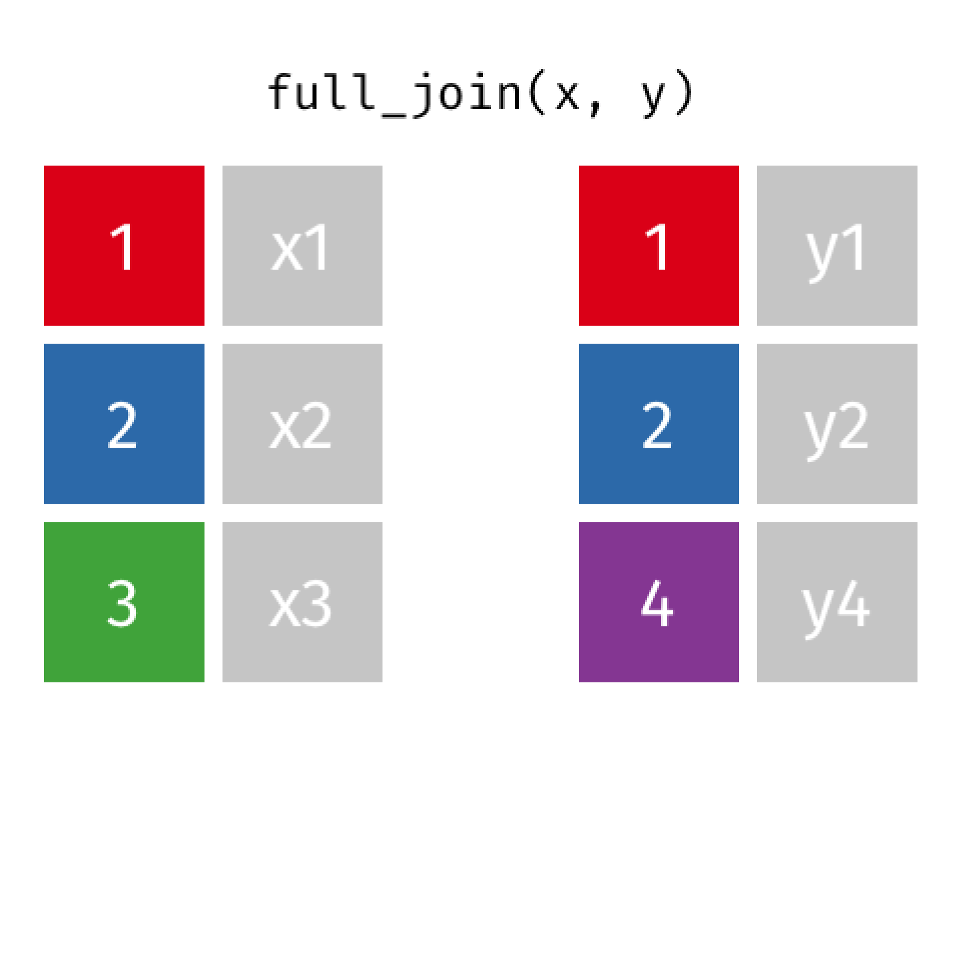
Right-Join

- Start with all rows from `y` and add all columns from `x`
  - Rows in `y` with no match in `x` (4) will have `NA`s
  - Rows in `x` with no match in `y` (3) will be dropped



Full-Join

- All rows and all columns from `x` and `y`
  - Rows that do not match (3 and 4) will have `NA`s



Joining Two *Different* Datasets: Overview

References

- `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]