Manipulating and Summarizing Data

Adam Kuczynski

UNIVERSITY OF WASHINGTON

Today's Theme:

"Data Engineer Work"

Issues around preparing a dataset for the analyses you want to run:

- Subsetting data
- Performing operations across rows and columns
- Creating new variables
- Creating rich summaries of your data
- Merging multiple datasets together

Tibbles

What is a `tibble`?

From the tibble webpage:

> "A **tibble**, or `tbl_df` [the official `class` of a tibble] is a modern reimagining of the data.frame, keeping what time has proben to be effective, and throwing out what is not. Tibbles are data.frames that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading to cleaner, more expressive code. Tibbles also have an enhanced print() method which makes them easier to use with large datasets containing complex objects."

Tibbles are part of the Tidyverse and operate in much the same way as dataframes (most of the time you don't need to worry about whether your object is a tibble or a dataframe)

Although most functions from the Tidyverse set of packages will operate on both dataframes and tibbles, some (e.g., `group_by()`) will return a tibble back

Making Tibbles

There are three primary ways to make a tibble:

1. Convert a dataframe into a tibble using `tibble::as_tibble()`[1]

```
mtcars <- as_tibble(mtcars)
class(mtcars)
```

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

2. Use `tibble::tibble()`

```
uwclinpsych <- tibble(name = c("Corey", "Angela", "Bill", "Mary", "Jane", "Lori"),
                      grads = c(1, 0, 4, 3, 2, 3),
                      fullprof = c(F, F, T, T, T, T))

head(uwclinpsych, 4)
```

```
## # A tibble: 4 x 3
##   name   grads fullprof
##   <chr>  <dbl> <lgl>
## 1 Corey      1 FALSE
## 2 Angela     0 FALSE
## 3 Bill       4 TRUE
## 4 Mary       3 TRUE
```

[1] The `as_tibble()` function has been exported into the `tidyr` and `dplyr` package, so loading any of those will give you access to this function.

3. use `tibble::tribble()` to construct a tibble row-wise. Column names are denoted with a `~` in front and are not quoted. Values are comma separated and rows are separated by a newline.

```r
uwclinpsych <- tribble(
  ~name,      ~grads,  ~fullprof,
  "Corey",    1,       FALSE,
  "Angela",   0,       FALSE,
  "Bill",     4,       TRUE,
  "Mary",     3,       TRUE,
  "Jane",     2,       TRUE,
  "Lori",     3,       TRUE
)

print(uwclinpsych)
```

```
## # A tibble: 6 x 3
##   name   grads fullprof
##   <chr>  <dbl> <lgl>
## 1 Corey      1 FALSE
## 2 Angela     0 FALSE
## 3 Bill       4 TRUE
## 4 Mary       3 TRUE
## 5 Jane       2 TRUE
## 6 Lori       3 TRUE
```

A Nice Feature of `tibble()`

One really nice feature of constructing tibbles from scratch is that you can build on columns dynamically:

`tibble()`

```
tibble(nums = 1:10,
       lets = letters[1:10],
       both = paste0(nums, lets))
```

```
## # A tibble: 10 x 3
##     nums lets  both
##    <int> <chr> <chr>
##  1     1 a     1a
##  2     2 b     2b
##  3     3 c     3c
##  4     4 d     4d
##  5     5 e     5e
##  6     6 f     6f
##  7     7 g     7g
##  8     8 h     8h
##  9     9 i     9i
## 10    10 j     10j
```

`data.frame()`

```
data.frame(nums = 1:10,
           lets = letters[1:10],
           both = paste0(nums, lets))
```

```
## Error in paste0(nums, lets): object
```

```
df <- data.frame(nums = 1:10,
                 lets = letters[1:10])
df$both = paste0(df$nums, df$lets)
print(df)
```

👈 same output as `tibble()` except the class is dataframe

Manipulating Data

Starwars Data

To demonstrate much of `dplyr`'s functionality, we will use the `starwars` data that is loaded with `dplyr` and originally from SWAPI (a Star Wars API)

```r
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or…
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2…
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.…
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N…
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "…
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",…
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, …
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",…
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini…
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T…
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma…
## $ films      <list> <"The Empire Strikes Back", "Revenge of the Sith", "Return…
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp…
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",…
```

A Brief Reminder About Pipes

The `dplyr` package uses **verbs** to name the functions within. As a result, they work very nicely with the pipe (`%>%`) syntax

```
take_these_data %>%
    do_first_thing(with = this_value) %>%
    do_next_thing(using = that_value) %>%
```

The LHS is passed as the *first argument* to the function on the RHS

You can reference the LHS with a `.` to use it in other places in the RHS function

```
take_these_data %>%
  do_first_thing(argument = "Value", with = .) %>%
  do_next_thing(using = that_value) %>%
```

`group_by()`

`group_by()` is a special function that controls the behavior of other functions as they operate on the data

It returns a tibble with the following classes: `grouped_df`, `tbl_df`, `tbl`, and `data.frame`

Most functions called on grouped data operate *within each group* rather than on the entire dataset

Data are typically grouped by variables that are characters, factors, or integers, not continuous data

For example, `group_by()` characters' `eye_color`

```
starwars_grouped <- starwars %>%
  group_by(eye_color)

class(starwars_grouped)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

Notice that this dataset has *exactly* the same data as the ungrouped version, except it now controls the output of other function calls

Not grouped

```
dim(starwars)
```

```
## [1] 87 14
```

Grouped

```
dim(starwars_grouped)
```

```
## [1] 87 14
```

To remove a grouping structure use the `ungroup()` function (if left blank, *all* grouping is removed, otherwise just the specified groups are ungrouped):

```
starwars_ungrouped <- starwars_grouped %>%
  ungroup()
```

`class(starwars_ungrouped)`

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

`group_by()` example

Mean mass by character gender:

```
##  feminine masculine      none 
##  54.68889 106.14694  48.00000
```

```
starwars %>%
  # Center mass by sample average
  mutate(mass_gmc = mass - mean(mass, na.rm = T)) %>%
  group_by(gender) %>%
  # Center mass by group average
  mutate(mass_pmc = mass - mean(mass, na.rm = T)) %>%
  select(name, gender, mass, mass_gmc, mass_pmc)
```

```
## # A tibble: 87 x 5
## # Groups:   gender [3]
##    name               gender     mass mass_gmc mass_pmc
##    <chr>              <chr>     <dbl>    <dbl>    <dbl>
##  1 Luke Skywalker     masculine    77    -20.3   -29.1 
##  2 C-3PO              masculine    75    -22.3   -31.1 
##  3 R2-D2              masculine    32    -65.3   -74.1 
##  4 Darth Vader        masculine   136     38.7    29.9 
##  5 Leia Organa        feminine     49    -48.3    -5.69
##  6 Owen Lars          masculine   120     22.7    13.9 
##  7 Beru Whitesun lars feminine     75    -22.3    20.3 
##  8 R5-D4              masculine    32    -65.3   -74.1 
##  9 Biggs Darklighter  masculine    84    -13.3   -22.1 
## 10 Obi-Wan Kenobi     masculine    77    -20.3   -29.1 
## # … with 77 more rows
```

Grouping Metadata

Sometimes it can be helpful for you to programmatically refer to your grouping structure. `dplyr` offers four functions that return grouping metadata to help with this task:

Use these as stand-alone function calls:

- `group_data()` returns a dataframe with integer vectors specifying the rows that belong to each group
- `group_indices()` returns an integer vector of the same length as `nrow(grouped_data)` specifying which group index a row belongs to
- `group_vars()` returns a character vector of the colnames used for grouping
- `group_size()` returns an integer vector of length `n_groups()` specifying the number of rows within each group
- `n_group()` returns an integer vector of length 1 with the number of groups in the data

Use these inside other `dplyr` functions:

- `cur_data()` refers to the current (ungrouped) data for the current group
- `cur_group()` refers to the current group value(s)
- `cur_group_id()` refers to the current group unique numeric identifier

`rowwise()`

`rowwise()` allows you to perform operations on data one row at a time (equivalent to `group_by()` each row or `for` looping down each row)

For example, to simulate normally distributed data with different parameters:

```
df <- tibble(x = runif(6), y = runif(6), z = runif(6))
```

```
df %>%
  mutate(m = mean(c(x, y, z)))
```

```
## # A tibble: 6 x 4
##       x      y     z     m
##   <dbl>  <dbl> <dbl> <dbl>
## 1 0.266 0.945  0.687 0.553
## 2 0.372 0.661  0.384 0.553
## 3 0.573 0.629  0.770 0.553
## 4 0.908 0.0618 0.498 0.553
## 5 0.202 0.206  0.718 0.553
## 6 0.898 0.177  0.992 0.553
```

```
df %>%
  rowwise() %>%
  mutate(m = mean(c(x, y, z)))
```

```
## # A tibble: 6 x 4
## # Rowwise:
##       x      y     z     m
##   <dbl>  <dbl> <dbl> <dbl>
## 1 0.266 0.945  0.687 0.632
## 2 0.372 0.661  0.384 0.472
## 3 0.573 0.629  0.770 0.657
## 4 0.908 0.0618 0.498 0.489
## 5 0.202 0.206  0.718 0.375
## 6 0.898 0.177  0.992 0.689
```

A vectorized version: `df %>% mutate(m = rowMeans(select(., x, y, z)))`

`filter()`

`filter()` is used to subset rows from a dataframe

Similar to `[x, ]` except that it drops NAs

```
filter(.data, ..., .preserve = FALSE)
```

- `.data` is the data to subset on
- `...` are the condition(s) that specify the subset
- `.preserve` controls the grouping of the returned dataframe[1]

[1] If `.data` is grouped and `filter()` reduces the number of groups available in the data, the grouping will be recalculated (i.e., number of groups reduced) based on the new data when `.preserve` is set to `FALSE`

`filter()` Example

```
starwars %>%
  filter(mass > mean(mass, na.rm = T))
```

```
## # A tibble: 10 x 14
##    name    height  mass hair_color   skin_color eye_color birth_year sex   gender
##    <chr>    <int> <dbl> <chr>        <chr>      <chr>          <dbl> <chr> <chr>
##  1 Darth …    202   136 none         white      yellow          41.9 male  mascu…
##  2 Owen L…    178   120 brown, grey light       blue            52   male  mascu…
##  3 Chewba…    228   112 brown        unknown    blue           200   male  mascu…
##  4 Jabba …    175  1358 <NA>         green-tan… orange         600   herm… mascu…
##  5 Jek To…    180   110 brown        fair       blue            NA   male  mascu…
##  6 IG-88      200   140 none         metal      red             15   none  mascu…
##  7 Bossk      190   113 none         green      red             53   male  mascu…
##  8 Dexter…    198   102 none         brown      yellow          NA   male  mascu…
##  9 Grievo…    216   159 none         brown, wh… green, y…       NA   male  mascu…
## 10 Tarfful    234   136 brown        brown      blue            NA   male  mascu…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Notice that you don't need to refer to the `mass` column as `starwars$mass` because `filter()` already knows the context from the `.data` argument.

👇 same as

```
starwars[starwars$mass > mean(starwars$mass, na.rm = T)]
```

`filter()` Multiple Conditions

`filter()` can handle multiple conditions upon which to subset your data

When you pass `filter()` multiple conditions, they are combined with the `&` operator

```
starwars %>%
  filter(mass > mean(mass, na.rm = T),
         eye_color %in% c("blue", "red"))
```

```
## # A tibble: 6 x 14
##   name      height  mass hair_color  skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
## 1 Owen La…     178   120 brown, grey light      blue              52 male  mascu…
## 2 Chewbac…     228   112 brown       unknown    blue             200 male  mascu…
## 3 Jek Ton…     180   110 brown       fair       blue              NA male  mascu…
## 4 IG-88        200   140 none        metal      red               15 none  mascu…
## 5 Bossk        190   113 none        green      red               53 male  mascu…
## 6 Tarfful      234   136 brown       brown      blue              NA male  mascu…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

To `filter()` using the or (`|`) operator, include it on one line as you would inside `[ ]` or `subset()`

```r
starwars %>%
  filter(mass > mean(mass, na.rm = T) | eye_color %in% c("blue", "r
```

```
## # A tibble: 28 x 14
##    name     height  mass hair_color   skin_color eye_color birth_ye
##    <chr>     <int> <dbl> <chr>        <chr>      <chr>          <db
##  1 Luke S…     172    77 blond        fair       blue             19
##  2 R2-D2        96    32 <NA>         white, bl… red              33
##  3 Darth …     202   136 none         white      yellow           41
##  4 Owen L…     178   120 brown, grey  light      blue             52
##  5 Beru W…     165    75 brown        light      blue             47
##  6 R5-D4        97    32 <NA>         white, red red              NA
##  7 Anakin…     188    84 blond        fair       blue             41
##  8 Wilhuf…     180    NA auburn, gr… fair       blue             64
##  9 Chewba…     228   112 brown        unknown    blue            200
## 10 Jabba …     175  1358 <NA>         green-tan… orange          600
## # … with 18 more rows, and 5 more variables: homeworld <chr>, spe
## #   films <list>, vehicles <list>, starships <list>
```

Filter by Group

`filter()` will operate on grouped data:

👇 filter the entire dataset to characters whose mass is greater than average

```
starwars %>%
  filter(mass > mean(mass, na.rm = T)) %>%
  dim()
```

```
## [1] 10 14
```

👇 filter the entire dataset to characters whose mass is greater than the average *within their eye color*

```
starwars %>%
  group_by(eye_color) %>%
  filter(mass > mean(mass, na.rm = T)) %>%
  dim()
```

```
## [1] 27 14
```

`slice()`

`slice()` allows you to subset rows using their integer locations

Similar to passing `[x, ]` a numeric vector

- `slice()` (positive integers keep rows, negative integers remove rows)
- `slice_head()`: keep the top `n` or `prop` (proportion) rows
- `slice_tail()`: keep the bottom `n` or `prop` (proportion) rows
- `slice_min(order_by = col)`: keeps rows where `col` is at its minimum (`col` is a column in the data)
- `slice_max(order_by = col)`: keeps rows where `col` is at its maximum (`col` is a column in the data)
- `slice_sample()`: randomly selects `n` or `prop` rows

`slice()` example

To keep the 2nd, 54th, and 83rd row in the data:

```
starwars %>%
  slice(2, 54, 83)
```

```
## # A tibble: 3 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 C-3PO        167    75 <NA>       gold       yellow           112 none  mascul…
## 2 Yarael …     264    NA none       white      yellow            NA male  mascul…
## 3 Rey           NA    NA brown      light      hazel             NA fema… femini…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

To remove rows 1, 3-53, 55-82, and 84-87:

```
starwars %>%
  slice(-1, -(3:53), -(55:82), -(84:87))
```

```
## # A tibble: 3 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 C-3PO        167    75 <NA>       gold       yellow           112 none  mascul…
## 2 Yarael …     264    NA none       white      yellow            NA male  mascul…
## 3 Rey           NA    NA brown      light      hazel             NA fema… femini…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

`slice()`: A Warning

The starwars data has 87 rows:

```
nrow(starwars)
```

```
## [1] 87
```

Warning: If you try to slice off rows that don't exist, R will not throw an error! It just doesn't subset those rows...

```
starwars %>%
  slice(5, 100)
```

```
## # A tibble: 1 x 14
##   name      height  mass hair_color skin_color eye_color birth_yea
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl
## 1 Leia Or…     150    49 brown      light      brown              1
## # … with 5 more variables: homeworld <chr>, species <chr>, films
## #   vehicles <list>, starships <list>
```

`slice()` by Group

If the `.data` are grouped, `slice()` will operate on each group of the data:

```
starwars %>%
  filter(eye_color %in% c("black", "blue", "brown")) %>%
  group_by(eye_color) %>%
  slice(3, 10)
```

```
## # A tibble: 6 x 14
## # Groups:   eye_color [3]
##   name      height  mass hair_color skin_color  eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
## 1 Gasgano      122    NA none       white, blue black             NA male  mascu…
## 2 BB8           NA    NA none       none        black             NA none  mascu…
## 3 Beru Wh…     165    75 brown      light       blue              47 fema… femin…
## 4 Qui-Gon…     193    89 brown      fair        blue              92 male  mascu…
## 5 Han Solo     180    80 brown      fair        brown             29 male  mascu…
## 6 Shmi Sk…     163    NA black      fair        brown             72 fema… femin…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

`select()`

`select()` is used to subset columns from a dataframe (similar to `[, x]`, but variable names don't need to be quoted or passed as a vector)

```
starwars %>%
  select(character, mass, skin_color)
```

- `:` is used to select a range of consecutive columns

```
starwars %>%
  select(birth_year:species)
```

- `!` is used to negate a selection of columns

```
starwars %>%
  select(!birth_year, !vehicles, !starships)
```

- `-` is also used to negate a selection of columns

```
starwars %>%
  select(-birth_year, -vehicles, -starships)
```

`select()` helpers

You can dynamically select columns using a variety of helper functions from tidyselect:

- `where()` allows you to select columns based on a function that returns a logical value

```
starwars %>%
  select(where(is.character)) %>%
  colnames()
```

```
## [1] "name"       "hair_color" "skin_color" "eye_color"  "sex"
## [6] "gender"     "homeworld"  "species"
```

- `starts_with()` selects columns that start with a specified prefix

```
starwars %>%
  select(starts_with("h")) %>%
  colnames()
```

```
## [1] "height"     "hair_color" "homeworld"
```

- `ends_with()` selects all columns that end with a specified suffix

```
starwars %>%
  select(ends_with("color")) %>%
  colnames()
```

```
## [1] "hair_color" "skin_color" "eye_color"
```

- `contains()` selects all columns that contain a string

```
starwars %>%
  select(contains("me")) %>%
  colnames()
```

```
## [1] "name"      "homeworld"
```

- `num_range()` matches a numerical range (e.g., `phq_1`, `phq_2`, `phq_3`, ...)

```
psych::bfi %>%
  select(num_range("A", 1:5)) %>%
  colnames()
```

```
## [1] "A1" "A2" "A3" "A4" "A5"
```

- `matches()` matches a regular expression

```
starwars %>%
  select(matches("^[ns]|s$")) %>%
  colnames()
```

```
## [1] "name"       "mass"       "skin_color" "sex"        "species"
## [6] "films"      "vehicles"   "starships"
```

- `all_of()` matches column names from a character vector

```r
hcols <- c("height", "hair_color", "homeworld")

starwars %>%
  select(all_of(hcols)) %>%
  colnames()
```

```
## [1] "height"     "hair_color" "homeworld"
```

- `any_of()` works the same way as `all_of()` except that no error is thrown for non-existent columns:

```r
hcols <- c("height", "hair_color", "homeworld", "Not_a_Column")
```

```r
starwars %>%
  select(any_of(hcols)) %>%
  colnames()
```

```
## [1] "height"     "hair_color" "homeworld"
```

```r
starwars %>%
  select(all_of(hcols)) %>%
  colnames()
```

```
## Error: Can't subset columns that don't ex
## x Column `Not_a_Column` doesn't exist.
```

- `everything()` matches all columns

```r
starwars %>%
  select(everything()) %>%
  dim()
```

```
## [1] 87 14
```

Column Renaming

You can rename variables while selecting them with `select()`:[1]

```
starwars %>%
  select(character = name, weight = mass, height) %>%
  colnames()
```

```
## [1] "character" "weight"    "height"
```

However it is more explicitly done with `rename()`:

```
starwars %>%
  rename(character = name, weight = mass) %>%
  select(character, weight, height) %>%
  colnames()
```

```
## [1] "character" "weight"    "height"
```

[1] Notice the syntax: `select(new = old)`

Creating New Columns: `mutate()`

`mutate()` allows you to create a new column of data or modify existing columns

```
starwars %>%
  mutate(bmi = mass / (height/100)^2) %>% # convert cm to m
  select(mass, height, bmi) %>%
  head(3)
```

```
## # A tibble: 3 x 3
##    mass height   bmi
##   <dbl>  <int> <dbl>
## 1    77    172  26.0
## 2    75    167  26.9
## 3    32     96  34.7
```

We can also convert height to meters first, then calculate BMI:

```
starwars %>%
  mutate(height = height / 100,
         bmi = mass / height^2) %>%
  select(mass, height, bmi) %>%
  head(3)
```

👈 multiple statements can be placed in the same `mutate()` call, and just like in `tibble()`, they build on each other dynamically

```
## # A tibble: 3 x 3
##    mass height   bmi
##   <dbl>  <dbl> <dbl>
## 1    77   1.72  26.0
## 2    75   1.67  26.9
## 3    32   0.96  34.7
```

`mutate(across())`

The helper functions discussed for `select()` can also be used with `mutate()` to modify/create several columns simultaneously, using the `across()` function

`across()` takes column specifiers (e.g., using the helper functions) and a function to apply to the relevant columns. This function needs to be an anonymous function using either:

- `function(args) instructions`
- `\(args) instructions` (base R lambda function)
- `~ instructions` (`purrr`-style lambda function, where the current column is referenced as `.x`)

Three ways to square values across all numeric columns:

```r
# Using function(args) instructions
starwars %>%
  mutate(across(where(is.numeric),
                function(x) x^2))
```

```r
# Using \(args) instructions (R 4.1+ only)
starwars %>%
  mutate(across(where(is.numeric),
                \(x) x^2))
```

```r
# Using dplyr ~ syntax (purrr-style lambda function)
starwars %>%
  mutate(across(where(is.numeric),
                ~ .x^2))
```

`mutate()` Example

Let's say we want to compute a sum score for all the `vehicles` and `starships` a character has piloted (much like creating a sum score across several items from a test). The `vehicles` and `starships` columns are both list columns where each element (i.e., cell) is a character vector of all the vehicles/starships the character has piloted

First we want to determine the length of each vector for both variables, which corresponds with the number of vehicles/starships the character has piloted:

```
starwars <- starwars %>%
  rowwise() %>%
  mutate(across(vehicles:starships,
                ~ length(.x),
                .names = "{.col}_n")) %>%
  ungroup() # Remove rowwise() grouping structure

starwars %>% select(matches("^vehicles|^starships")) %>% head(4)
```

```
## # A tibble: 4 x 4
##   vehicles  starships vehicles_n starships_n
##   <list>    <list>         <int>       <int>
## 1 <chr [2]> <chr [2]>          2           2
## 2 <chr [0]> <chr [0]>          0           0
## 3 <chr [0]> <chr [0]>          0           0
## 4 <chr [0]> <chr [1]>          0           1
```

Then we want to take the sum of `vehicles_n` and `starships_n` and store it in a new variable called `total_piloted`:

```
starwars <- starwars %>%
  rowwise() %>%
  mutate(total_piloted = sum(vehicles_n, starships_n, na.rm = T)) %>%
  ungroup()

starwars %>% select(vehicles_n, starships_n, total_piloted) %>% head(4)
```

```
## # A tibble: 4 x 3
##   vehicles_n starships_n total_piloted
##        <int>       <int>         <int>
## 1          2           2             4
## 2          0           0             0
## 3          0           0             0
## 4          0           1             1
```

This can also be done without using `rowwise()` (it is very slow with large dataframes) using the vectorized `rowSums()` function:

```
starwars <- starwars %>%
  mutate(total_piloted = rowSums(select(., vehicles_n, starships_n),
                                 na.rm = T))
```

☝ `rowSums()` takes a dataframe as its first argument, so you need to use `select()` on `.`

if_else()

Remember `if_else()`? We can use it inside mutate:

```
starwars <- starwars %>%
  mutate(height_ordinal = if_else(height > (mean(height, na.rm = T) + sd(height, na
                                 if_else(height < (mean(height, na.rm = T) - sd(he
                                         "average")))

starwars %>% select(starts_with("height")) %>% slice_sample(n = 10)
```

```
## # A tibble: 10 x 2
##    height height_ordinal
##     <int> <chr>
##  1     NA <NA>
##  2    188 average
##  3    213 tall
##  4    178 average
##  5    191 average
##  6     97 short
##  7    170 average
##  8    178 average
##  9    163 average
## 10    196 average
```

`case_when()`

`case_when()` allows you to vectorize multiple `if_else()` statements:

```r
starwars <- starwars %>%
  mutate(height_ordinal = case_when(height > mean(height, na.rm = T) + sd(height,
                                    height < mean(height, na.rm = T) - sd(height,
                                    is.na(height) ~ NA_character_, # need to handl
                                    TRUE ~ "average")) # if none of the above are

starwars %>% select(starts_with("height")) %>% slice_sample(n = 10)
```

```
## # A tibble: 10 x 2
##    height height_ordinal
##     <int> <chr>
##  1     NA <NA>
##  2    188 average
##  3    213 tall
##  4    178 average
##  5    191 average
##  6     97 short
##  7    170 average
##  8    178 average
##  9    163 average
## 10    196 average
```

Example Data

Consider the following example data from a fake EMA study (2 participants, 2 days, 3x/day):

```
## # A tibble: 12 x 5
##        id   day  ping      x       y
##     <dbl> <dbl> <int>  <dbl>   <dbl>
##  1   1001     1     1  1.35  -0.163
##  2   1001     1     2  9.21   1.73
##  3   1001     1     3  1.99   0.0638
##  4   1001     2     1 -0.759 -1.21
##  5   1001     2     2  3.86   1.98
##  6   1001     2     3  0.863 -0.129
##  7   1002     1     1  4.89   2.35
##  8   1002     1     2  3.53  -1.12
##  9   1002     1     3  0.539  1.64
## 10   1002     2     1  1.10   1.21
## 11   1002     2     2  9.48   5.13
## 12   1002     2     3  6.30   2.44
```

```
d <- tibble(id = rep(c(1001, 1002), ea
            day = rep(c(1, 1, 1, 2, 2,
            ping = rep(1:3, 4),
            x = rnorm(12, 5, 4),
            y = x*0.3 + rnorm(12))
```

- `id` = participant ID
- `day` = day in study
- `ping` = prompt within each day
- `x` = predictor variable
- `y` = outcome variable

`lag()` and `lead()`

`lag()` allows you to create lagged variables for analysis (`lead()` does the opposite):

```
d <- d %>%
  mutate(x_l1 = lag(x), # lag(1)
         x_l2 = lag(x, 2)) # lag(2)

print(d)
```

```
## # A tibble: 12 x 7
##        id   day  ping      x       y   x_l1   x_l2
##     <dbl> <dbl> <int>  <dbl>   <dbl>  <dbl>  <dbl>
##  1   1001     1     1  1.35  -0.163  NA     NA
##  2   1001     1     2  9.21   1.73    1.35  NA
##  3   1001     1     3  1.99   0.0638  9.21   1.35
##  4   1001     2     1 -0.759 -1.21    1.99   9.21
##  5   1001     2     2  3.86   1.98   -0.759  1.99
##  6   1001     2     3  0.863 -0.129   3.86  -0.759
##  7   1002     1     1  4.89   2.35    0.863  3.86
##  8   1002     1     2  3.53  -1.12    4.89   0.863
##  9   1002     1     3  0.539  1.64    3.53   4.89
## 10   1002     2     1  1.10   1.21    0.539  3.53
## 11   1002     2     2  9.48   5.13    1.10   0.539
## 12   1002     2     3  6.30   2.44    9.48   1.10
```

Targeting the `nth()` Elements

It is often useful to target the nth element of a vector when manipulating your data. For this, we can use the `first()`, `last()`, and `nth()` functions. For example, let's say we want to predict `y` with lag(1) `x`, but we don't want to use the last observation of the day to predict the first of the next day. We can make all relevant values NA for the analysis:

```
d %>%
  group_by(id, day) %>%
  arrange(id, day, ping) %>% # Necessary to make sure last() works
  mutate(x_l1_NAlast = case_when(ping == last(ping) ~ NA_real_,
                                 TRUE ~ x_l1))
```

```
## # A tibble: 12 x 8
## # Groups:   id, day [4]
##       id   day  ping      x       y   x_l1   x_l2 x_l1_NAlast
##    <dbl> <dbl> <int>  <dbl>   <dbl>  <dbl>  <dbl>       <dbl>
##  1  1001     1     1  1.35  -0.163  NA     NA          NA
##  2  1001     1     2  9.21   1.73    1.35  NA           1.35
##  3  1001     1     3  1.99   0.0638  9.21   1.35       NA
##  4  1001     2     1 -0.759 -1.21    1.99   9.21        1.99
##  5  1001     2     2  3.86   1.98   -0.759  1.99       -0.759
##  6  1001     2     3  0.863 -0.129   3.86  -0.759      NA
##  7  1002     1     1  4.89   2.35    0.863  3.86        0.863
##  8  1002     1     2  3.53  -1.12    4.89   0.863       4.89
##  9  1002     1     3  0.539  1.64    3.53   4.89       NA
## 10  1002     2     1  1.10   1.21    0.539  3.53        0.539
## 11  1002     2     2  9.48   5.13    1.10   0.539       1.10
## 12  1002     2     3  6.30   2.44    9.48   1.10       NA
```

`arrange()`

`arrange()` orders the rows of a dataframe by values within the specified columns

Values are arranged in ascending order my default. To arrange by a column in descending order, use the `desc()` function:

```
# Arrange ascending by id, then day, then descending by ping
d %>%
  arrange(id, day, desc(ping))
```

```
## # A tibble: 12 x 7
##       id   day  ping      x       y   x_l1   x_l2
##    <dbl> <dbl> <int>  <dbl>   <dbl>  <dbl>  <dbl>
##  1  1001     1     3  1.99   0.0638  9.21   1.35
##  2  1001     1     2  9.21   1.73    1.35  NA
##  3  1001     1     1  1.35  -0.163  NA     NA
##  4  1001     2     3  0.863 -0.129   3.86  -0.759
##  5  1001     2     2  3.86   1.98   -0.759  1.99
##  6  1001     2     1 -0.759 -1.21    1.99   9.21
##  7  1002     1     3  0.539  1.64    3.53   4.89
##  8  1002     1     2  3.53  -1.12    4.89   0.863
##  9  1002     1     1  4.89   2.35    0.863  3.86
## 10  1002     2     3  6.30   2.44    9.48   1.10
## 11  1002     2     2  9.48   5.13    1.10   0.539
## 12  1002     2     1  1.10   1.21    0.539  3.53
```

`coalesce()`

Sometimes you have two or more mutually exclusive variables that belong in the same column for analysis. `coalesce()` helps you combines these variables by finding the first non-`NA` value.

For example, suppose you have 3 columns representing how much participants like their Windows, Mac, or Linux computer (depending on which operating system they use), but you only care about their computer rating:

```r
d_comp <- tibble(id = 1001:1003,
                 Windows = c(NA, NA, 3),
                 Mac = c(7, NA, NA),
                 Linux = c(NA, 10, NA))

print(d_comp)
```

```
## # A tibble: 3 x 4
##      id Windows   Mac Linux
##   <int>   <dbl> <dbl> <dbl>
## 1  1001      NA     7    NA
## 2  1002      NA    NA    10
## 3  1003       3    NA    NA
```

Using `coalesce()` we can combine these into one variable for analysis:

```r
d_comp <- d_comp %>%
  mutate(rating = coalesce(Windows, Mac, Linux)) %>%
  select(id, rating)
```

```
## # A tibble: 3 x 5
##      id  rating
##   <int>   <dbl>
## 1  1001       7
## 2  1002      10
## 3  1003       3
```

`na_if()`

`na_if()` allows you to replace specified values with `NA`

```r
starwars %>%
  mutate(eye_color_cleaned = na_if(eye_color, "unknown")) %>%
  select(name, eye_color, eye_color_cleaned) %>%
  tail(5)
```

```
## # A tibble: 5 x 3
##   name           eye_color eye_color_cleaned
##   <chr>          <chr>     <chr>
## 1 Rey            hazel     hazel
## 2 Poe Dameron    brown     brown
## 3 BB8            black     black
## 4 Captain Phasma unknown   <NA>
## 5 Padmé Amidala  brown     brown
```

This does the same exact thing as:

```r
dataframe$column[dataframe$column == val] <- NA
```

```r
dataframe %>%
  mutate(column = if_else(column == value, NA, column))
```

```r
dataframe %>%
  mutate(column = case_when(column == value ~ NA,
                            TRUE ~ column))
```

`relocate()`

`relocate()` allows you to rearrange columns in a dataframe

```
relocate(.data, ..., .before = NULL, .after = NULL)
```

- `.data` is the dataframe to reorder
- `...` are the columns to move
- `.before` destination to move columns before (colname, index, `tidy-select`)
- `.after` destination to move columns after (colname, index, `tidy-select`)

For example, to move all the numeric columns before the character columns:

```
starwars %>%
  relocate(where(is.numeric), .before = where(is.character)) %>%
  sapply(class)
```

```
##          height           mass     birth_year     vehicles_n    starships_n
##       "integer"      "numeric"      "numeric"      "integer"      "integer"
##  total_piloted           name     hair_color     skin_color      eye_color
##       "integer"    "character"    "character"    "character"    "character"
##             sex         gender      homeworld        species          films
##     "character"    "character"    "character"    "character"         "list"
##        vehicles      starships height_ordinal
##          "list"         "list"    "character"
```

`distinct()`

`distinct()` selects the unique rows from a data.frame (similar to the `unique.data.frame()` function in base R, but it is faster when you are working with a large dataframe). For example:

```r
df <- tibble(a = c(1, 1, 2, 2),
             b = c(1, 1, 2, 1),
             c = c(3, 3, 2, 3))
```

```
## # A tibble: 4 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     1     3
## 2     1     1     3
## 3     2     2     2
## 4     2     1     3
```

```r
# Find distinct rows based on cols a and b
df %>%
  distinct(a, b)
```

```
## # A tibble: 3 x 2
##       a     b
##   <dbl> <dbl>
## 1     1     1
## 2     2     2
## 3     2     1
```

```r
# To keep all columns
df %>%
  distinct(a, b, .keep_all = TRUE)
```

```
## # A tibble: 3 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     1     3
## 2     2     2     2
## 3     2     1     3
```

`pull()`

`pull()` allows you to extract a column from a dataframe as a vector and is equivalent to `$`

```
d$ping
```

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

```
d %>% pull(ping)
```

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

`pull()` is useful when you want a column after manipulating a dataframe. For example, find the mean height of characters whose mass is greater than average and who are not orifinally from the planet Naboo:

```
starwars %>%
  filter(mass > mean(mass, na.rm = T),
         homeworld != "Naboo") %>%
  pull(height) %>%
  mean()
```

```
## [1] 200.1111
```

Summarizing Data

`summarize()`

`summarize()` returns a dataframe with specified summary statistic(s) of your data with 1+ rows for each combination of grouping variables (1 for no grouping structure) and 1 column for each summary statistic (much like `tapply()`, but much more flexible with cleaner output)

```r
mtcars %>%
  summarize(nobs = n(),
            mpg_mean = mean(mpg, na.rm = T),
            mpg_sd = sd(mpg, na.rm = T))
```

```
##   nobs mpg_mean   mpg_sd
## 1   32 20.09062 6.026948
```

When we group the data before calling `summarize()` we will get summary statistics for each group:

```r
mtcars %>%
  group_by(cyl) %>%
  summarize(nobs = n(),
            mpg_mean = mean(mpg, na.rm = T),
            mpg_sd = sd(mpg, na.rm = T))
```

```
## # A tibble: 3 x 4
##     cyl  nobs mpg_mean mpg_sd
##   <dbl> <int>    <dbl>  <dbl>
## 1     4    11     26.7   4.51
## 2     6     7     19.7   1.45
## 3     8    14     15.1   2.56
```

`summarize()` whatever you want

You can specify any function you want within `summarize()`

```
mtcars %>%
  group_by(cyl) %>%
  summarize(nobs = n(), # n observations for each group
            percentN = (n() / nrow(.)) * 100, # percent of total observations
            from_total_obs = nrow(.) - n(), # distance from total observations
            mpg_sd_sample = sd(mpg, na.rm = T), # sample SD (n-1 in denominator)
            mpg_sd_pop = sqrt(sum((mpg - mean(mpg, na.rm = T))^2) / (n()))) # popu
```

```
## # A tibble: 3 x 6
##     cyl  nobs percentN from_total_obs mpg_sd_sample mpg_sd_pop
##   <dbl> <int>    <dbl>          <int>         <dbl>      <dbl>
## 1     4    11     34.4             21          4.51       4.30
## 2     6     7     21.9             25          1.45       1.35
## 3     8    14     43.8             18          2.56       2.47
```

`summarize(across())`

Much like `mutate(across())` allowed us to manipulate several columns at once, `summarize(across())` allows us to programmatically target columns for summarizing

For example, to take the mean across all numeric variables:

```
starwars %>%
  group_by(sex) %>%
  summarize(across(where(is.numeric),
                   ~ mean(.x, na.rm = T),
                   .names = "mean_{.col}"))
```

```
## # A tibble: 5 x 7
##   sex     mean_height mean_mass mean_birth_year mean_vehicles_n mean_starships_n
##   <chr>         <dbl>     <dbl>           <dbl>           <dbl>            <dbl>
## 1 female         169.      54.7            47.2           0.125            0.188
## 2 hermap…        175     1358             600             0                0
## 3 male           179.      81.0            85.5           0.183            0.45
## 4 none           131.      69.8            53.3           0                0
## 5 <NA>           181.      48              62             0                0.25
## # … with 1 more variable: mean_total_piloted <dbl>
```

`summarize(across())` with Multiple Summary Functions

To summarize across columns with more than one summary function (e.g., count, mean, and sd), used a named list of summary functions:

```
starwars %>%
  group_by(sex) %>%
  summarize(across(where(is.numeric),
                   list(nobs = ~ sum(!is.na(.x)), mean = mean, sd = sd), # list of
                   na.rm = T)) # additional arguments passed to the functions
```

```
## # A tibble: 5 x 19
##   sex            height_nobs height_mean height_sd mass_nobs mass_mean mass_sd
##   <chr>                <int>       <dbl>     <dbl>     <int>     <dbl>   <dbl>
## 1 female                  15        169.     15.3          9      54.7    8.59
## 2 hermaphroditic           1        175      NA            1    1358     NA
## 3 male                    57        179.     36.0         44      81.0   28.2
## 4 none                     5        131.     49.1          4      69.8   51.0
## 5 <NA>                     3        181.      2.89         1      48     NA
## # … with 12 more variables: birth_year_nobs <int>, birth_year_mean <dbl>,
## #   birth_year_sd <dbl>, vehicles_n_nobs <int>, vehicles_n_mean <dbl>,
## #   vehicles_n_sd <dbl>, starships_n_nobs <int>, starships_n_mean <dbl>,
## #   starships_n_sd <dbl>, total_piloted_nobs <int>, total_piloted_mean <dbl>,
## #   total_piloted_sd <dbl>
```

Merging Data

Merging Dataframes

Merging dataframes together is an essential part of data management. For example:

- Merging baseline data with EMA data
- Merging self-report data to physiological data
- Merging patient data from multiple clinics
- Merging departmental revenue data to employee wage statistics

Sometimes marging data is a large part of the entire project, as in the Washington Merged Longitudinal Administrative Data project at UW, which takes data from several WA State agencies to answer novel administrative questions.

Questions to Ask Yourself When Merging

When merging datasets `A` and `B`:

- Which *rows* do you want to keep from each dataframe?
- Which *columns* do you want to keep from each dataframe?
- Which variable(s) determine whether rows *match*?

Data Example for Merging

To keep things simple, let's use the following data to practice merging:

```r
A <- tibble(ID = 0:6,
            x = rnorm(7, c(2, 2, 2, 2,
            y = x*0.5 + rnorm(7))
```

```
## # A tibble: 7 x 3
##      ID     x     y
##   <int> <dbl> <dbl>
## 1     0  1.37 1.43
## 2     1  2.18 1.67
## 3     2  1.16 0.277
## 4     3  3.60 3.31
## 5     4  5.33 3.05
## 6     5  4.18 1.47
## 7     6  5.49 0.529
```

```r
B <- tibble(ID = 1:7,
            group = c(rep(1:2, each =
            age = sample(20:50, 7))
```

```
## # A tibble: 7 x 3
##      ID group   age
##   <int> <dbl> <int>
## 1     1     1    24
## 2     2     1    21
## 3     3     1    29
## 4     4     2    44
## 5     5     2    31
## 6     6     2    34
## 7     7     2    20
```

Notice that `ID == 0` is not in `B` and `ID == 7` is not in `A`

We will use the `ID` column to merge the data (in the `by =` argument)

`left_join()`

`left_join(A, B)` keeps all rows from `A`, all cols from `A` and `B`

`dplyr`

```
left_join(A, B, by = "ID")
```

```
## # A tibble: 7 x 5
##       ID     x     y group   age
##    <int> <dbl> <dbl> <dbl> <int>
## 1      0  1.37 1.43     NA    NA
## 2      1  2.18 1.67      1    24
## 3      2  1.16 0.277     1    21
## 4      3  3.60 3.31      1    29
## 5      4  5.33 3.05      2    44
## 6      5  4.18 1.47      2    31
## 7      6  5.49 0.529     2    34
```

☝️ `ID == 7` from `B` not in merged data ☝️

Base R Equivalent

```
merge(A, B, by = "ID", all.x = T)
```

```
##   ID        x         y group age
## 1  0 1.373546 1.4250978    NA  NA
## 2  1 2.183643 1.6676030     1  24
## 3  2 1.164371 0.2767973     1  21
## 4  3 3.595281 3.3094216     1  29
## 5  4 5.329508 3.0545971     2  44
## 6  5 4.179532 1.4685252     2  31
## 7  6 5.487429 0.5290146     2  34
```

Most of your joins will probably be `left_join()`

`right_join()`

`right_join(A, B)` keeps all rows from `B`, all cols from `A` and `B`

`dplyr`

```
right_join(A, B, by = "ID")
```

```
## # A tibble: 7 x 5
##      ID     x     y group   age
##   <int> <dbl> <dbl> <dbl> <int>
## 1     1  2.18 1.67      1    24
## 2     2  1.16 0.277     1    21
## 3     3  3.60 3.31      1    29
## 4     4  5.33 3.05      2    44
## 5     5  4.18 1.47      2    31
## 6     6  5.49 0.529     2    34
## 7     7 NA   NA         2    20
```

☝ `ID == 0` from `A` not in merged data ☝

Base R Equivalent

```
merge(A, B, by = "ID", all.y = T)
```

```
##   ID        x         y group age
## 1  1 2.183643 1.6676030     1  24
## 2  2 1.164371 0.2767973     1  21
## 3  3 3.595281 3.3094216     1  29
## 4  4 5.329508 3.0545971     2  44
## 5  5 4.179532 1.4685252     2  31
## 6  6 5.487429 0.5290146     2  34
## 7  7       NA        NA     2  20
```

`inner_join()`

`inner_join(A, B)` keeps only rows from `A` and `B` that match, all cols from `A` and `B`

`dplyr`

```
inner_join(A, B, by = "ID")
```

```
## # A tibble: 6 x 5
##      ID     x     y group   age
##   <int> <dbl> <dbl> <dbl> <int>
## 1     1  2.18 1.67      1    24
## 2     2  1.16 0.277     1    21
## 3     3  3.60 3.31      1    29
## 4     4  5.33 3.05      2    44
## 5     5  4.18 1.47      2    31
## 6     6  5.49 0.529     2    34
```

Base R Equivalent

```
merge(A, B, by = "ID", all = F)
```

```
##   ID        x         y group age
## 1  1 2.183643 1.6676030     1  24
## 2  2 1.164371 0.2767973     1  21
## 3  3 3.595281 3.3094216     1  29
## 4  4 5.329508 3.0545971     2  44
## 5  5 4.179532 1.4685252     2  31
## 6  6 5.487429 0.5290146     2  34
```

☝️ Neither `ID == 7` from `A` nor `ID == 0` from `B` in merged data ☝️

`all = F` is equivalent to `all.x = F, all.y = F` in base R's `merge()`

`full_join()`

`full_join(A, B)` keeps rows from both `A` and `B`, all cols from `A` and `B`

`dplyr`

```r
full_join(A, B, by = "ID")
```

```
## # A tibble: 8 x 5
##        ID     x       y group   age
##     <int> <dbl>   <dbl> <dbl> <int>
## 1       0  1.37  1.43      NA    NA
## 2       1  2.18  1.67       1    24
## 3       2  1.16  0.277      1    21
## 4       3  3.60  3.31       1    29
## 5       4  5.33  3.05       2    44
## 6       5  4.18  1.47       2    31
## 7       6  5.49  0.529      2    34
## 8       7 NA    NA          2    20
```

Base R Equivalent

```r
merge(A, B, by = "ID", all = T)
```

```
##   ID        x         y group age
## 1  0 1.373546 1.4250978    NA  NA
## 2  1 2.183643 1.6676030     1  24
## 3  2 1.164371 0.2767973     1  21
## 4  3 3.595281 3.3094216     1  29
## 5  4 5.329508 3.0545971     2  44
## 6  5 4.179532 1.4685252     2  31
## 7  6 5.487429 0.5290146     2  34
## 8  7       NA        NA     2  20
```

☝️ `ID == 7` from `A` and `ID == 0` from `B` are in merged data ☝️

`all = T` is equivalent to `all.x = T, all.y = T` in base R's `merge()`

`semi_join()`

`semi_join(A, B)` keeps only rows from both `A` and `B` that match, only cols from `A`

`dplyr`

```r
semi_join(A, B, by = "ID")
```

```
## # A tibble: 6 x 3
##      ID     x     y
##   <int> <dbl> <dbl>
## 1     1  2.18 1.67
## 2     2  1.16 0.277
## 3     3  3.60 3.31
## 4     4  5.33 3.05
## 5     5  4.18 1.47
## 6     6  5.49 0.529
```

Base R Equivalent

```r
merge(A, B[, colnames(B) %in% colnames
      by = "ID", all = F)
```

```
##   ID        x         y
## 1  1 2.183643 1.6676030
## 2  2 1.164371 0.2767973
## 3  3 3.595281 3.3094216
## 4  4 5.329508 3.0545971
## 5  5 4.179532 1.4685252
## 6  6 5.487429 0.5290146
```

`anti_join()`

`anti_join(A, B)` keeps rows from `A` that *do not* match `B`, only cols from `A`

`dplyr`

```
anti_join(A, B, by = "ID")
```

```
## # A tibble: 1 x 3
##      ID     x     y
##   <int> <dbl> <dbl>
## 1     0  1.37  1.43
```

Base R Equivalent

```
merge(A[!A$ID %in% B$ID, ],
      B[, colnames(B) %in% colnames(A)],
      by = "ID", all.x = T)
```

```
##   ID        x        y
## 1  0 1.373546 1.425098
```

`all = T` is equivalent to `all.x = T, all.y = T` in base R's `merge()`

Matching With `by =`

- You can pass `by =` a character vector of columns upon which to match. This is useful when merging nested data (e.g., data from clinic visits nested within patients)

```r
left_join(A, B, by = c("ID", "Date"))
merge(A, B, by = c("ID", "Date"), all.x = T)
```

- If the `by` columns used for merging don't have the same name (e.g., "PatientNum" and "PatientID"):

```r
left_join(A, B, by = c("PatientNum" = "PatientID"))
merge(A, B, by.x = "PatientNum", by.y = "PatientID", all.x = T)
```