Joins and logic

Week 4

AEM 2850 / 5850 : R for Business Analytics
Cornell Dyson
Fall 2025

Acknowledgements: Grant McDermott, Jenny Bryan, R4DS (2e), Garrick Aden-Buie

Announcements

Reminders:

- Submit assignments via canvas / gradescope
    - Homework - Week 3 was due yesterday (Monday) at 11:59pm

Questions before we get started?

Plan for this week

Tuesday

Prologue

Joins

example-04-1

Thursday

Logic

- Boolean algebra
- Conditional transformations

example-04-2

Prologue

What sports do we watch?

Take a guess: what's the most popular spectator sport among classmates?

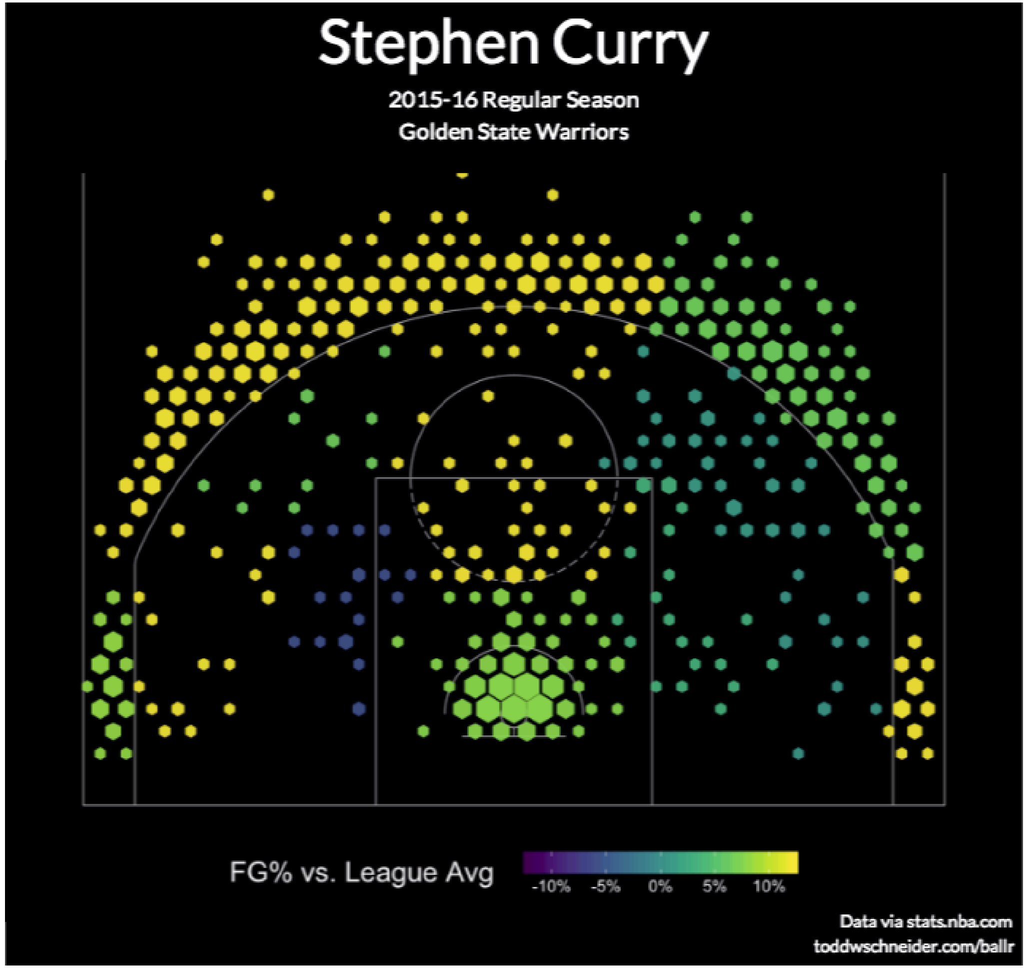
Here are the first 20 responses:

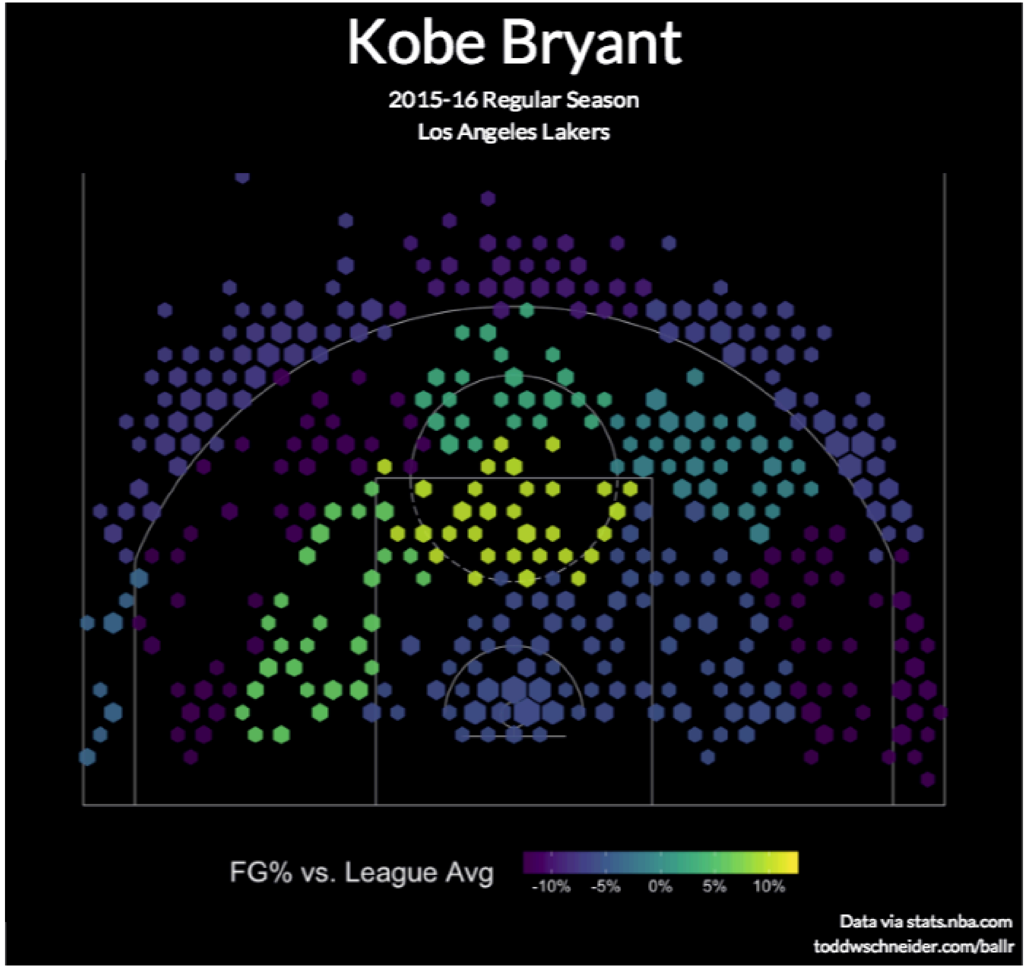
```
##  [1] "soccer"     "baseball"   "baseball"   "badminton"  "football"
##  [6] "baseball"   "volleyball" "swimming"   "football"   "tennis"
## [11] "soccer"     "football"   "soccer"     "volleyball" "basketball"
## [16] "hockey"     "tennis"     "volleyball" "baseball"   "soccer"
```

Let's `count` and `arrange` to get the top 3:

```
## # A tibble: 3 × 2
##   sport          n
##   <chr>      <int>
## 1 basketball    14
## 2 baseball      12
## 3 soccer        12
```

R can be used for sports analytics, too!





Joins

Joins

Most data analyses require information contained in multiple data frames

We **join** them together to answer questions

Keys are the variables that connect a pair of data frames in a join

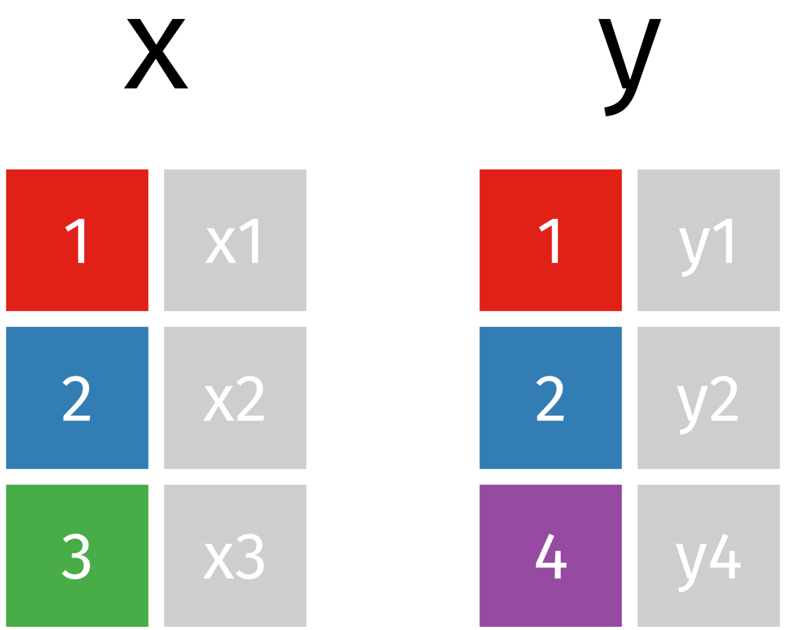
Join verbs from dplyr

1. **Mutating joins**: add new variables
    - `left_join()`
    - `right_join()`
    - `inner_join()`
    - `full_join()`
2. **Filtering joins**: filter observations
    - `semi_join()`
    - `anti_join()`

Join animations

Let's start by visualizing joins

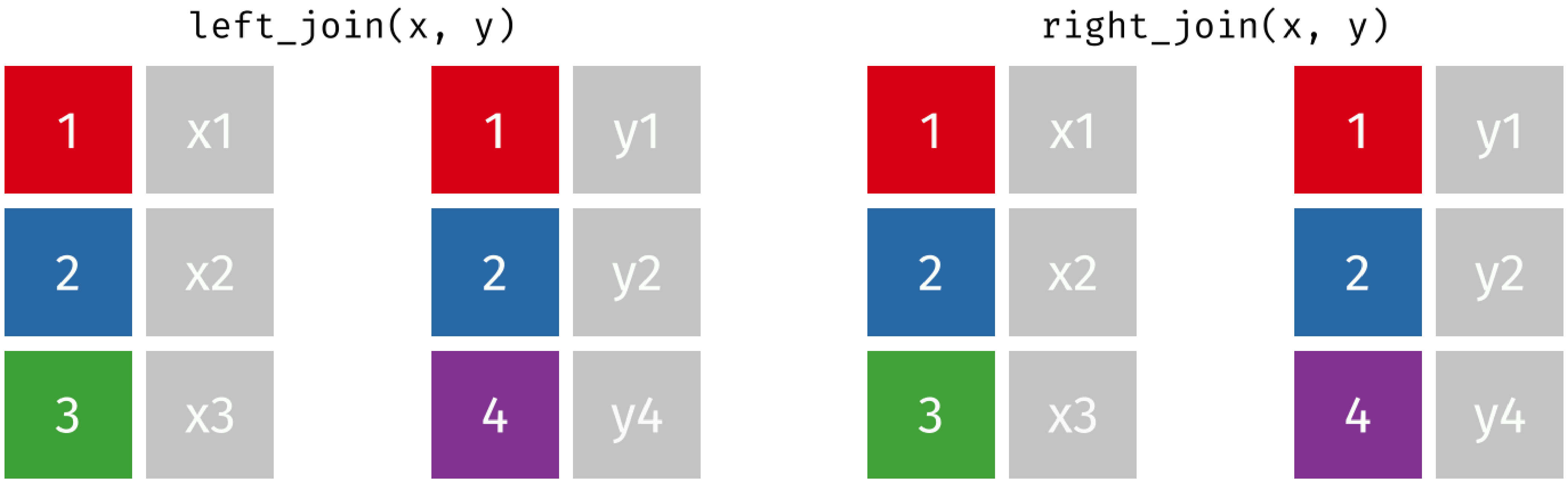
Here are two data frames we want to **join**



Their **keys** are in color in the first column, and other data are in grey

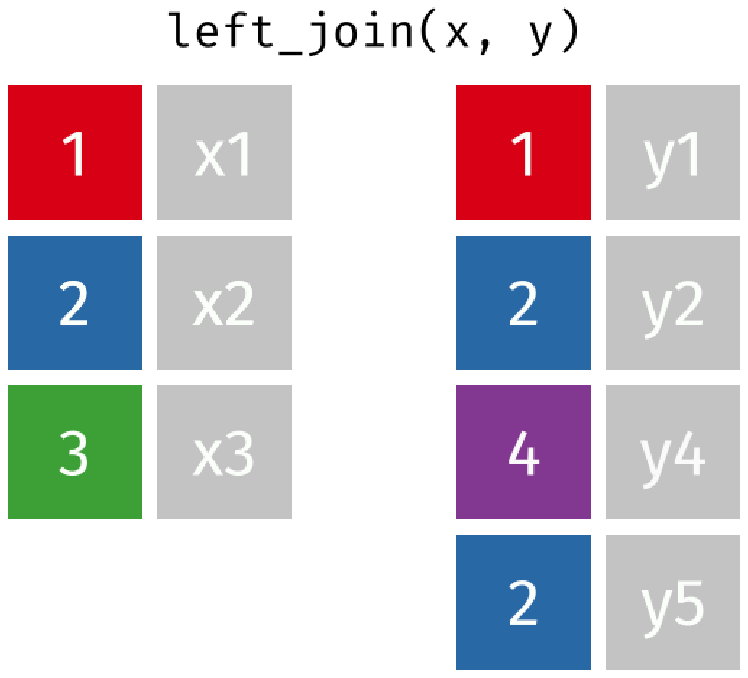
Left join and right join

Left or right joins add variables to the left or right data frames



Multiple matches

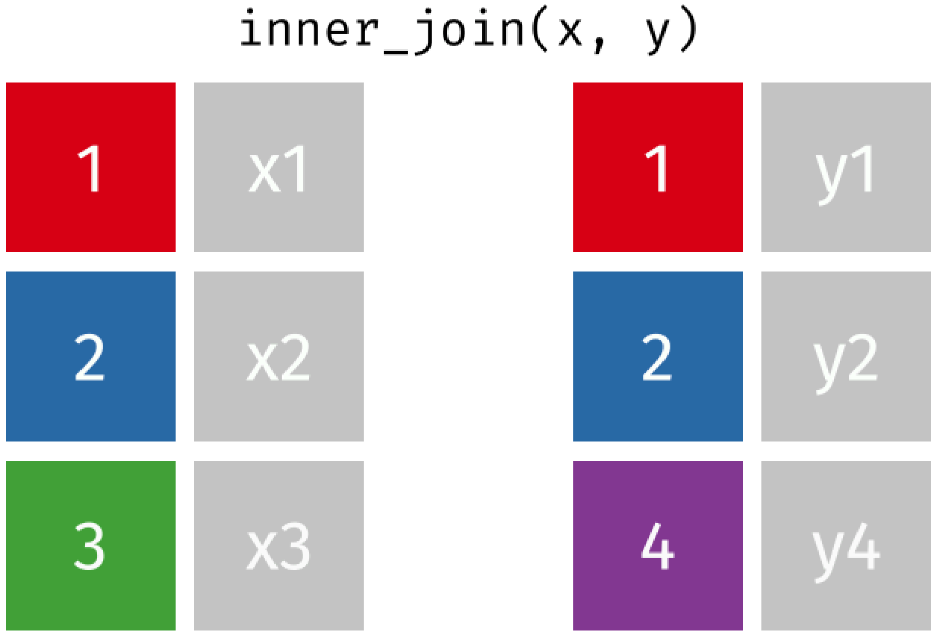
With multiple matches between x and y, all combinations of matches are returned



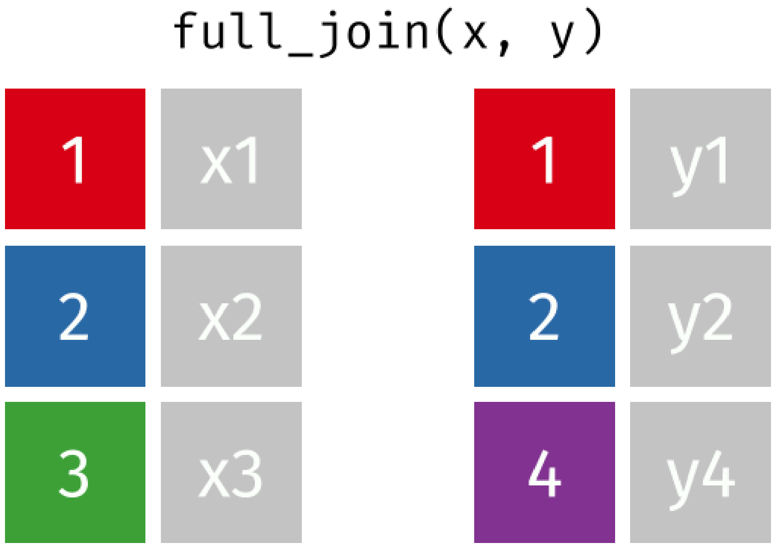
In this example, x2 is duplicated to join one row in x to multiple rows in y

Inner join and full join

Inner joins return all rows in x **AND** y



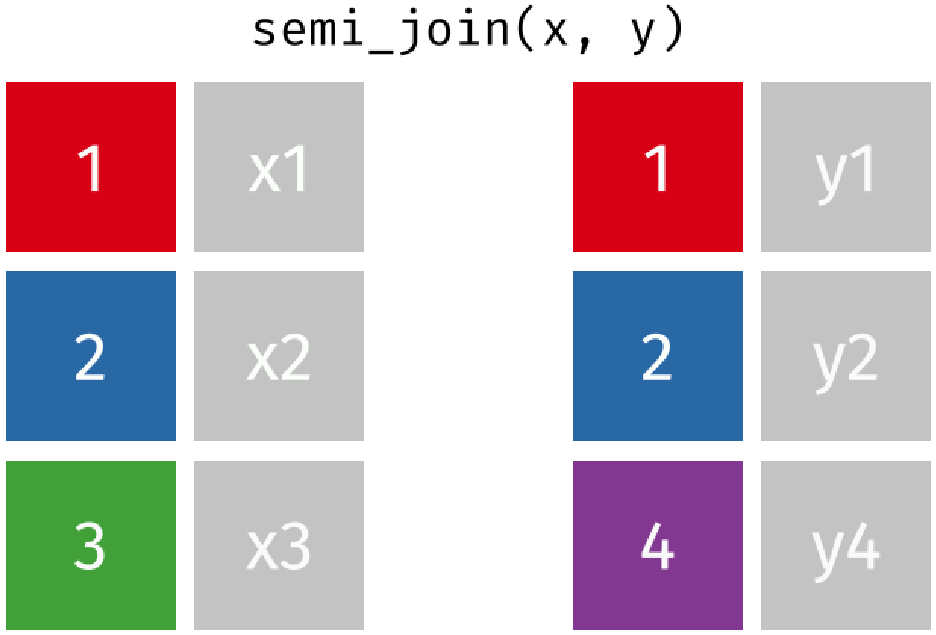
Full joins return all rows in x **OR** y

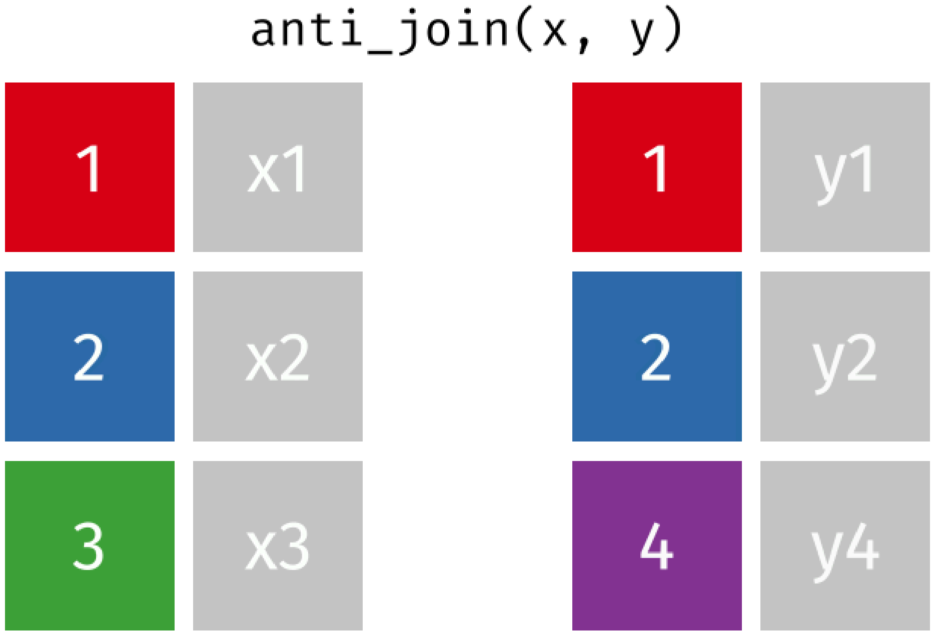


Semi join and anti join

Semi joins filter rows in x that match y

Anti joins filter rows in x **not** in y





example-04-1

Additional slides on joins for your reference

Joins

Let's learn these join commands using two small data frames

```
superheroes
```

```
## # A tibble: 7 × 3
##   name      alignment publisher
##   <chr>     <chr>     <chr>
## 1 Magneto   bad       Marvel
## 2 Storm     good      Marvel
## 3 Mystique  bad       Marvel
## 4 Batman    good      DC
## 5 Joker     bad       DC
## 6 Catwoman  bad       DC
## 7 Hellboy   good      Dark Horse Comics
```

```
publishers
```

```
## # A tibble: 3 × 2
##   publisher year_founded
##   <chr>            <int>
## 1 DC                1934
## 2 Marvel            1939
## 3 Image             1992
```

1) dplyr::left_join(x, y)

```
left_join(superheroes, publishers)
```

```
## Joining with `by = join_by(publisher)`
```

```
## # A tibble: 7 × 4
##   name      alignment publisher         year_founded
##   <chr>     <chr>     <chr>                    <int>
## 1 Magneto   bad       Marvel                    1939
## 2 Storm     good      Marvel                    1939
## 3 Mystique  bad       Marvel                    1939
## 4 Batman    good      DC                        1934
## 5 Joker     bad       DC                        1934
## 6 Catwoman  bad       DC                        1934
## 7 Hellboy   good      Dark Horse Comics           NA
```

`left_join` is a **mutating join**: it adds variables to `x`

`left_join` returns all rows from `x`

2) dplyr::right_join(x, y)

```
right_join(superheroes, publishers)
```

```
## Joining with `by = join_by(publisher)`
```

```
## # A tibble: 7 × 4
##   name      alignment publisher year_founded
##   <chr>     <chr>     <chr>            <int>
## 1 Magneto   bad       Marvel            1939
## 2 Storm     good      Marvel            1939
## 3 Mystique  bad       Marvel            1939
## 4 Batman    good      DC                1934
## 5 Joker     bad       DC                1934
## 6 Catwoman  bad       DC                1934
## 7 <NA>      <NA>      Image             1992
```

`right_join` is a **mutating join**: it adds variables to `y`

`right_join` returns all rows from `y`

3) dplyr::inner_join(x, y)

```
inner_join(superheroes, publishers)
```

```
## Joining with `by = join_by(publisher)`
```

```
## # A tibble: 6 × 4
##   name      alignment publisher year_founded
##   <chr>     <chr>     <chr>            <int>
## 1 Magneto   bad       Marvel            1939
## 2 Storm     good      Marvel            1939
## 3 Mystique  bad       Marvel            1939
## 4 Batman    good      DC                1934
## 5 Joker     bad       DC                1934
## 6 Catwoman  bad       DC                1934
```

How is `inner_join` different from `left_join` and `right_join`?

`inner_join` returns all rows in x **AND** y

4) dplyr::full_join(x, y)

```r
full_join(superheroes, publishers) # how many rows do you think this will produce?
```

```
## Joining with `by = join_by(publisher)`
```

```
## # A tibble: 8 × 4
##   name      alignment publisher         year_founded
##   <chr>     <chr>     <chr>                    <int>
## 1 Magneto   bad       Marvel                    1939
## 2 Storm     good      Marvel                    1939
## 3 Mystique  bad       Marvel                    1939
## 4 Batman    good      DC                        1934
## 5 Joker     bad       DC                        1934
## 6 Catwoman  bad       DC                        1934
## 7 Hellboy   good      Dark Horse Comics           NA
## 8 <NA>      <NA>      Image                     1992
```

`full_join` returns all rows in x **OR** y

5) dplyr::semi_join(x, y)

```
superheroes
```

```
## # A tibble: 7 × 3
##   name      alignment publisher
##   <chr>     <chr>     <chr>
## 1 Magneto   bad       Marvel
## 2 Storm     good      Marvel
## 3 Mystique  bad       Marvel
## 4 Batman    good      DC
## 5 Joker     bad       DC
## 6 Catwoman  bad       DC
## 7 Hellboy   good      Dark Horse Comics
```

```
semi_join(superheroes, publishers)
```

```
## Joining with `by = join_by(publisher)`

## # A tibble: 6 × 3
##   name      alignment publisher
##   <chr>     <chr>     <chr>
## 1 Magneto   bad       Marvel
## 2 Storm     good      Marvel
## 3 Mystique  bad       Marvel
## 4 Batman    good      DC
## 5 Joker     bad       DC
## 6 Catwoman  bad       DC
```

`semi_join` is a **filtering join**: it keeps observations in `x` that have a match in `y`

Note that the variables do not change

6) dplyr::anti_join(x, y)

```
superheroes
```

```
## # A tibble: 7 × 3
##   name      alignment publisher
##   <chr>     <chr>     <chr>
## 1 Magneto   bad       Marvel
## 2 Storm     good      Marvel
## 3 Mystique  bad       Marvel
## 4 Batman    good      DC
## 5 Joker     bad       DC
## 6 Catwoman  bad       DC
## 7 Hellboy   good      Dark Horse Comics
```

```
anti_join(superheroes, publishers)
```

```
## Joining with `by = join_by(publisher)`

## # A tibble: 1 × 3
##   name    alignment publisher
##   <chr>   <chr>     <chr>
## 1 Hellboy good      Dark Horse Comics
```

`anti_join` is a **filtering join**: it keeps obs. in `x` that **DO NOT** have a match in `y`

Note that the variables do not change

Key variables

How do `dplyr` join commands know what variables to use as **keys**?

By default, `*_join()` uses all variables that are common across `x` and `y`

```
intersect(names(superheroes), names(publishers)) # variable used for matching before
```

```
## [1] "publisher"
```

Or, we can specify what to join by: `*_join(..., by = join_by(publisher))`

Note: before `dplyr` 1.1.0, the syntax was: `*_join(..., by = "publisher")`

Exploring keys

```
library(nycflights13) # let's explore keys using the nycflights13 data
flights |> print(n = 8) # print the first 8 rows of flights
```

```
## # A tibble: 336,776 × 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## 7  2013     1     1      555            600        -5      913            854
## 8  2013     1     1      557            600        -3      709            723
## # ℹ 336,768 more rows
## # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exploring keys

```
planes # print the first 10 rows of planes
```

```
## # A tibble: 3,322 × 9
##    tailnum  year type                   manufacturer model engines seats speed engine
##    <chr>   <int> <chr>                  <chr>        <chr>   <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing multi… EMBRAER      EMB-…       2    55    NA Turbo…
##  2 N102UW   1998 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  3 N103US   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  4 N104UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  5 N10575   2002 Fixed wing multi… EMBRAER      EMB-…       2    55    NA Turbo…
##  6 N105UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  7 N107US   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  8 N108UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  9 N109UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
## 10 N110UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
## # ℹ 3,312 more rows
```

Let's perform a left join on flights and planes

```
left_join(flights, planes) |>
  select(year:dep_time, arr_time, carrier:tailnum, type, model) |> # keep text to one slide
  print(n = 5) # just to save vertical space on the slide
```

```
## Joining with `by = join_by(year, tailnum)`
```

```
## # A tibble: 336,776 × 10
##    year month   day dep_time arr_time carrier flight tailnum type  model
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1  2013     1     1      517      830 UA        1545 N14228  <NA>  <NA>
## 2  2013     1     1      533      850 UA        1714 N24211  <NA>  <NA>
## 3  2013     1     1      542      923 AA        1141 N619AA  <NA>  <NA>
## 4  2013     1     1      544     1004 B6         725 N804JB  <NA>  <NA>
## 5  2013     1     1      554      812 DL         461 N668DN  <NA>  <NA>
## # ℹ 336,771 more rows
```

Uh-oh! What's up with `type` and `model`?

Uh-oh!

As before, `dplyr` guessed which columns to join on

It uses columns with the same name:

```
## Joining, by = c("year", "tailnum")
```

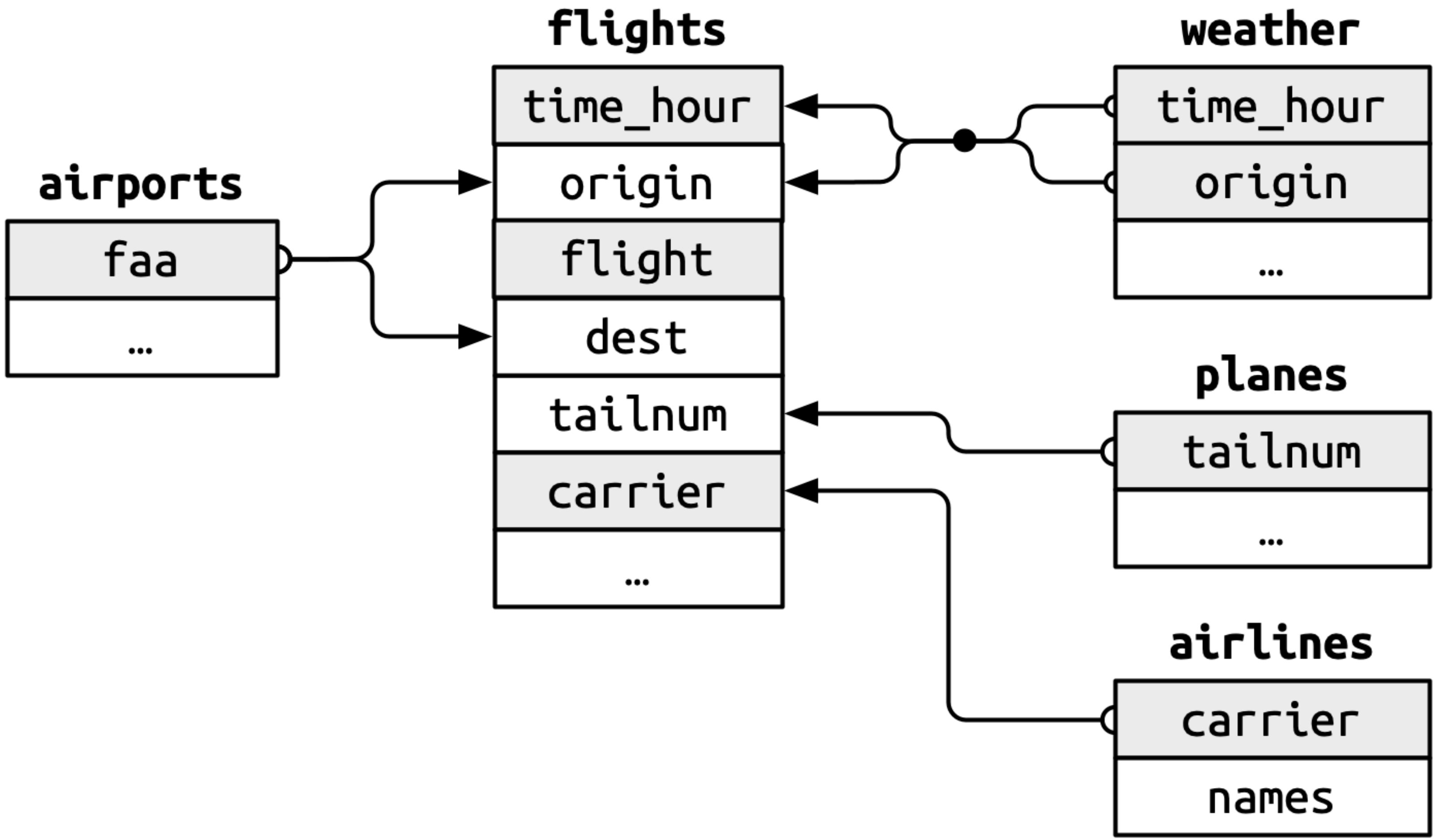
Does anyone see a potential problem here?

The variable `year` does not have a consistent meaning across the datasets

In `flights` it refers to the *year of flight*, in `planes` it refers to *year of construction*

Luckily we can avoid this by using the argument `by = join_by(...)`

What should we join flights and planes by?



Specifying join keys

We just need to be explicit in the join call by using the by argument

```
left_join(flights,
          planes |> rename(year_built = year), # not necessary w/ below line, but helpful
          by = join_by(tailnum) # be specific about the joining column
          ) |>
  select(year, month:dep_time, carrier, flight, tailnum, year_built, type, model) |>
  print(n = 5) # just to save vertical space on the slide
```

```
## # A tibble: 336,776 × 10
##    year month   day dep_time carrier flight tailnum year_built type        model
##   <int> <int> <int>    <int> <chr>    <int> <chr>        <int> <chr>       <chr>
## 1  2013     1     1      517 UA        1545 N14228        1999 Fixed wing… 737-…
## 2  2013     1     1      533 UA        1714 N24211        1998 Fixed wing… 737-…
## 3  2013     1     1      542 AA        1141 N619AA        1990 Fixed wing… 757-…
## 4  2013     1     1      544 B6         725 N804JB        2012 Fixed wing… A320…
## 5  2013     1     1      554 DL         461 N668DN        1991 Fixed wing… 757-…
## # ℹ 336,771 more rows
```

Specifying join keys

What happens if we don't rename `year` before this join?

```
left_join(flights,
          planes, # not renaming "year" to "year_built" this time
          by = join_by(tailnum)
          ) |>
  select(contains("year"), month:dep_time, arr_time, carrier, flight, tailnum, type, model) |>
  print(n = 4) # just to save vertical space on the slide
```

```
## # A tibble: 336,776 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type  model
##    <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1   2013   1999     1     1      517      830 UA        1545 N14228  Fixe… 737-…
## 2   2013   1998     1     1      533      850 UA        1714 N24211  Fixe… 737-…
## 3   2013   1990     1     1      542      923 AA        1141 N619AA  Fixe… 757-…
## 4   2013   2012     1     1      544     1004 B6         725 N804JB  Fixe… A320…
## # ℹ 336,772 more rows
```

What is `year.x`? What is `year.y`?

Summary of key verbs so far

Key verbs

Import

readr

1. `read_csv`
2. `write_csv`

readxl

1. `read_excel`

Tidy

tidyr

1. `pivot_longer`
2. `pivot_wider`
3. `separate_wider_delim`

Join

dplyr

1. `left_join`
2. `right_join`
3. `inner_join`
4. `full_join`
5. `semi_join`
6. `anti_join`

Transform

dplyr

1. `filter`
2. `arrange`
3. `select`
4. `mutate`
5. `summarize`

Logic

Logical vectors

What values can the logical data type take?

Logical values can be `TRUE`, `FALSE`, or `NA`

What are **logical vectors**?

Logical vectors are just vectors ("columns") that only contain `TRUE`, `FALSE`, or `NA`

Logical vectors

Can you think of any logical vectors we have worked with so far?

While we don't often see logical vectors in raw data, we use them all the time!

Example: every time we make comparisons to `filter()` data we create *transient* logical variables that are computed, used, and then thrown away

Transient logical vectors

We create a transient logical vector when we filter `flights` to Miami:

```r
library(nycflights13)
flights |>
  select(carrier, flight, dest) |>
  filter(dest == "MIA")
```

```
## # A tibble: 11,728 × 3
##    carrier flight dest
##    <chr>    <int> <chr>
##  1 AA        1141 MIA
##  2 AA        1895 MIA
##  3 UA        1077 MIA
##  4 AA        1837 MIA
##  5 DL        2003 MIA
##  6 AA        2279 MIA
##  7 AA        2267 MIA
##  8 DL        1843 MIA
##  9 AA         443 MIA
## 10 DL        2143 MIA
## # ℹ 11,718 more rows
```

`filter(dest == "MIA")`: under the hood

```r
# create a logical vector from a comparison
flights |>
  select(carrier, flight, dest) |>
  mutate(welcome_to_miami = dest == "MIA")
```

```
## # A tibble: 336,776 × 4
##    carrier flight dest  welcome_to_miami
##    <chr>    <int> <chr> <lgl>
##  1 UA        1545 IAH   FALSE
##  2 UA        1714 IAH   FALSE
##  3 AA        1141 MIA   TRUE
##  4 B6         725 BQN   FALSE
##  5 DL         461 ATL   FALSE
##  6 UA        1696 ORD   FALSE
##  7 B6         507 FLL   FALSE
##  8 EV        5708 IAD   FALSE
##  9 B6          79 MCO   FALSE
## 10 AA         301 ORD   FALSE
## # ℹ 336,766 more rows
```

```r
flights |>
  select(carrier, flight, dest) |>
  mutate(welcome_to_miami = dest == "MIA") |>
  filter(welcome_to_miami) # then filter
```

```
## # A tibble: 11,728 × 4
##    carrier flight dest  welcome_to_miami
##    <chr>    <int> <chr> <lgl>
##  1 AA        1141 MIA   TRUE
##  2 AA        1895 MIA   TRUE
##  3 UA        1077 MIA   TRUE
##  4 AA        1837 MIA   TRUE
##  5 DL        2003 MIA   TRUE
##  6 AA        2279 MIA   TRUE
##  7 AA        2267 MIA   TRUE
##  8 DL        1843 MIA   TRUE
##  9 AA         443 MIA   TRUE
## 10 DL        2143 MIA   TRUE
## # ℹ 11,718 more rows
```

Comparisons

Numeric comparisons like `<`, `<=`, `>`, `>=`, `!=`, and `==` can be used to create logical vectors

As we have seen, `==` and `!=` are useful for comparing characters (i.e., strings)

`is.na()`

`is.na()` is a useful function for checking whether something is `NA`

Why use `is.na(x)` when we could just use `x == NA`?

```
x <- 2850 + 5850
is.na(x)
```

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

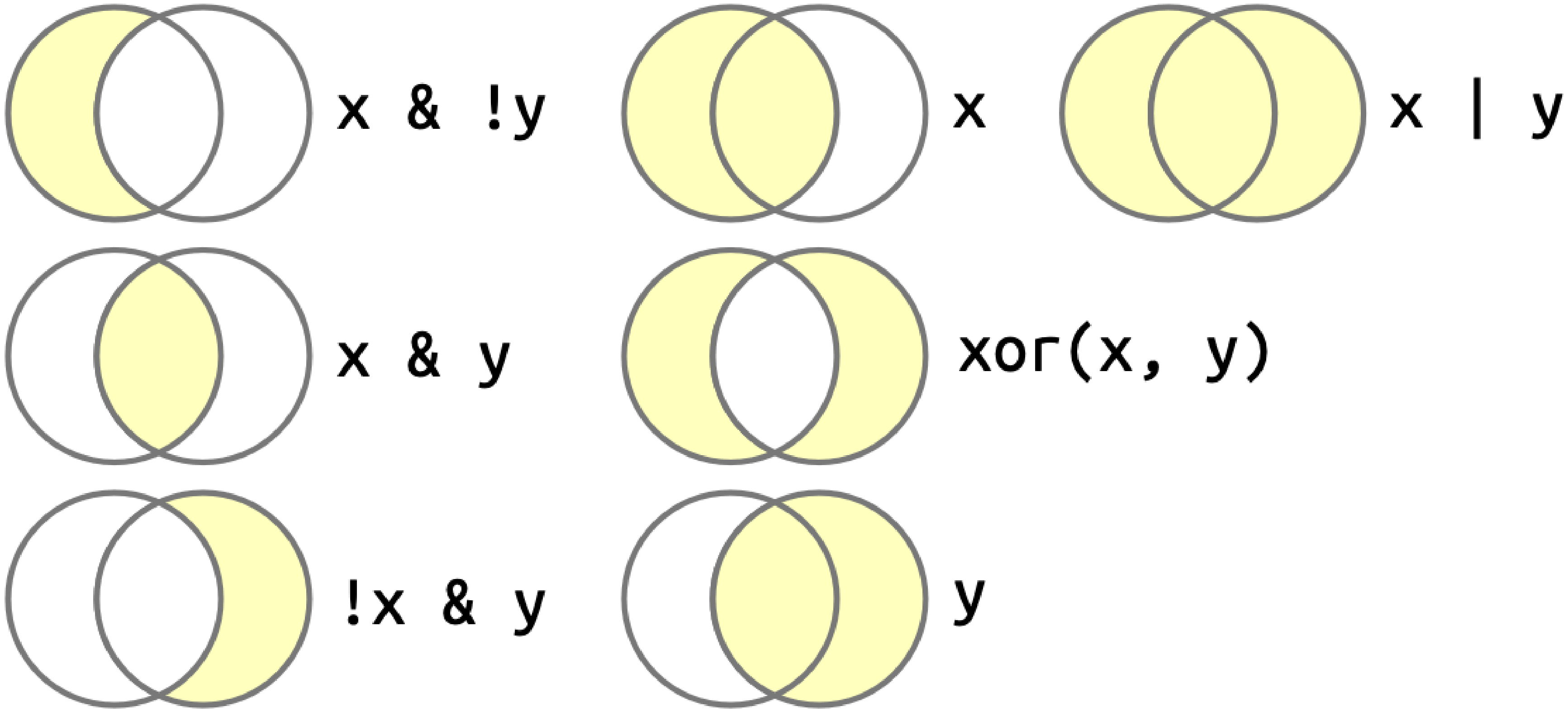
```
x == NA
```

```
## [1] NA
```

May seem odd but it makes sense when you think about concrete comparisons

Boolean algebra

Use Boolean algebra to combine comparisons / logical vectors



Boolean operator examples

```r
flights |>
  select(carrier, flight, dest) |>
  filter(dest == "MIA" | dest == "MYR")
```

```
## # A tibble: 11,787 × 3
##    carrier flight dest
##    <chr>    <int> <chr>
##  1 AA        1141 MIA
##  2 AA        1895 MIA
##  3 UA        1077 MIA
##  4 AA        1837 MIA
##  5 DL        2003 MIA
##  6 AA        2279 MIA
##  7 AA        2267 MIA
##  8 DL        1843 MIA
##  9 AA         443 MIA
## 10 EV        4412 MYR
## # ℹ 11,777 more rows
```

```r
flights |>
  select(carrier, flight, dest) |>
  filter(carrier == "AA" & dest == "MIA")
```

```
## # A tibble: 7,234 × 3
##    carrier flight dest
##    <chr>    <int> <chr>
##  1 AA        1141 MIA
##  2 AA        1895 MIA
##  3 AA        1837 MIA
##  4 AA        2279 MIA
##  5 AA        2267 MIA
##  6 AA         443 MIA
##  7 AA         647 MIA
##  8 AA        2099 MIA
##  9 AA        1623 MIA
## 10 AA        2253 MIA
## # ℹ 7,224 more rows
```

%in%

`x %in% y` is a useful shortcut for identifying whether a value in `x` is contained in `y`

```
flights |>
  select(carrier, flight, dest) |>
  filter(dest %in% c("MIA", "MYR"))
```

```
## # A tibble: 11,787 × 3
##    carrier flight dest
##    <chr>    <int> <chr>
##  1 AA        1141 MIA
##  2 AA        1895 MIA
##  3 UA        1077 MIA
##  4 AA        1837 MIA
##  5 DL        2003 MIA
##  6 AA        2279 MIA
##  7 AA        2267 MIA
##  8 DL        1843 MIA
##  9 AA         443 MIA
## 10 EV        4412 MYR
## # ℹ 11,777 more rows
```

Numeric operations on logical vectors

Numeric operations treat `TRUE` as `1` and `FALSE` as `0`:

```
x <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
```

```
sum(x)
```

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

```
mean(x)
```

```
## [1] 0.4
```

```
min(x)
```

```
## [1] 0
```

```
max(x)
```

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

This can be handy when doing calculations that depend on conditions

Conditional transformations

`if_else()` can be used to do things based on a binary condition

```
flights |> filter(dest == "MIA") |>
    select(carrier, flight, dest, sched_dep_time) |>
    mutate(too_early = if_else(sched_dep_time < 800, "too early!", "okay"))
```

```
## # A tibble: 11,728 × 5
##    carrier flight dest  sched_dep_time too_early
##    <chr>    <int> <chr>          <int> <chr>
##  1 AA        1141 MIA              540 too early!
##  2 AA        1895 MIA              610 too early!
##  3 UA        1077 MIA              607 too early!
##  4 AA        1837 MIA              610 too early!
##  5 DL        2003 MIA              700 too early!
##  6 AA        2279 MIA              700 too early!
##  7 AA        2267 MIA              755 too early!
##  8 DL        1843 MIA              800 okay
##  9 AA         443 MIA              715 too early!
## 10 DL        2143 MIA              900 okay
## # ℹ 11,718 more rows
```

Conditional transformations

`case_when()` is a more flexible approach that allows many different conditions

```
flights |>
  filter(dest == "MIA") |>
  select(carrier, flight, sched_dep_time) |>
  mutate(too_early = case_when(
    sched_dep_time < 600   ~ "too early!",
    sched_dep_time < 800   ~ "still early",
    sched_dep_time <= 2000 ~ "okay",
    sched_dep_time > 2000  ~ "late"
    )
  )
```

```
## # A tibble: 11,728 × 4
##    carrier flight sched_dep_time too_early
##    <chr>    <int>          <int> <chr>
##  1 AA        1141            540 too early!
##  2 AA        1895            610 still early
##  3 UA        1077            607 still early
##  4 AA        1837            610 still early
##  5 DL        2003            700 still early
##  6 AA        2279            700 still early
##  7 AA        2267            755 still early
##  8 DL        1843            800 okay
##  9 AA         443            715 still early
## 10 DL        2143            900 okay
## # ℹ 11,718 more rows
```

Conditions are evaluated in order

`condition ~ output` syntax is new; watch out for overlapping conditions!

example-04-2