Wrangling Notes - Chapter 4, 5 and 6

Wrangling in the Tidyverse (Ch 4)

Key wrangling verbs are introduced:

- select()
- filter()
- mutate()
- arrange()
- summarize()
- group_by()
- rename()

Demos functions commonly used in summarize

- min, max
- sum
- mean
- n()
- pmin, pmax

You might also be interested in:

- sd
- median

Other commands/concepts demo-ed (Ch 4)

- ==, | , and & as the equality, `or`, and `and` operators
- lubridate package for working with dates
- %>% - piping operator
- %in% operator
- skim()
- ifelse(), including nested ifelse statements
- case_when() expression

You might also be interested in:

- != as the not equals operator
- can do - inside select to remove variables (if keeping many and removing few)

Joins (Ch 5)

Key join concepts are introduced including:

- inner_join
- left_join
- right_join

You might also be interested in:

- full_join

Other commands/concepts demo-ed (Ch 5)

- paste
- n_distinct
- concept of foreign `keys` relating tables / data sets
- na.rm = TRUE

Tidying Data (Ch 6)

In tidy data, 2 rules are followed. First, the rows (observations) refer to a specific, unique, and similar sort of "thing" or object. Second, the columns (variables) each have the same sort of value recorded for each row. I.E. data is not tidy if you have both county and state level summary information in your rows (the rows represent different types of objects and some entries would be aggregates other others).

This chapter demos using googlesheets4 including read_sheet and unnest() and shows two other key functions for reshaping data:

- pivot_longer
- pivot_wider

Other commands/concepts demo-ed (Ch 6)

- collapse()
- parse_number()
- nest()
- map()
- pull()
- pluck()
- Naming and code style conventions
- Case matters
- janitor package including clean_names()

Other functions of interest

This list is other functions that may or may not be demo-ed in the text (I tried to note highlights above), but that can be useful when wrangling.

- ungroup()
- na.omit()
- names_glue option within pivot_wider()
- relocate()
- unite()

You should know how to look these up now to learn about what they do.

Data

We return to our FIFA 2019 data to illustrate some of the wrangling commands and concepts. (A separate dataset is used below to illustrate working with dates.)

```
Fifadata <- read_csv("https://awagaman.people.amherst.edu/stat240/FifaData2019Subset.csv")
```

```
## Rows: 18207 Columns: 40
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (4): Name, Club, PreferredFoot, Position
## dbl (36): Age, Overall, Crossing, Finishing, HeadingAccuracy, ShortPassing, ...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

For our visualizations, we used only players with an overall rating of over 85. We also only used a few variables. The full data set is somewhat large - 18207 observations and 40 variables.

```
dim(Fifadata)
```

```
## [1] 18207    40
```

```
names(Fifadata)
```

```
##  [1] "Name"            "Age"             "Overall"         "Club"
##  [5] "PreferredFoot"   "Position"        "Crossing"        "Finishing"
##  [9] "HeadingAccuracy" "ShortPassing"    "Volleys"         "Dribbling"
## [13] "Curve"           "FKAccuracy"      "LongPassing"     "BallControl"
## [17] "Acceleration"    "SprintSpeed"     "Agility"         "Reactions"
## [21] "Balance"         "ShotPower"       "Jumping"         "Stamina"
## [25] "Strength"        "LongShots"       "Aggression"      "Interceptions"
## [29] "Positioning"     "Vision"          "Penalties"       "Composure"
## [33] "Marking"         "StandingTackle"  "SlidingTackle"   "GKDiving"
## [37] "GKHandling"      "GKKicking"       "GKPositioning"   "GKReflexes"
```

For our visualizations previously, we didn't select the variables to reduce the number in the data set (I was trying not to introduce too many commands at once). But we can if we want to make it easier to work with. For our visuals, we needed the following variables:

- Overall
- PreferredFoot
- Position
- FKAccuracy
- Dribbling

So, we only need those 5 variables for players with overall ratings > 85. We might also want to keep Name just in case we see outliers and want to know who they are. What verbs should we use to generate this new data set?

```
Fifasmall <- Fifadata %>%
  filter(Overall > 85) %>%
  select(Name, Overall, PreferredFoot, Position, FKAccuracy, Dribbling)

glimpse(Fifasmall)
```

```
## Rows: 77
## Columns: 6
## $ Name          <chr> "L. Messi", "Cristiano Ronaldo", "Neymar Jr", "De Gea", ~
## $ Overall       <dbl> 94, 94, 92, 91, 91, 91, 91, 91, 91, 90, 90, 90, 90, 90, ~
## $ PreferredFoot <chr> "Left", "Right", "Right", "Right", "Right", "Right", "Ri~
## $ Position      <chr> "RF", "ST", "LW", "GK", "RCM", "LF", "RCM", "RS", "RCB",~
## $ FKAccuracy    <dbl> 94, 76, 87, 19, 83, 79, 78, 84, 72, 14, 86, 84, 51, 77, ~
```

```
## $ Dribbling     <dbl> 97, 88, 96, 18, 86, 95, 90, 87, 63, 12, 85, 81, 53, 89, ~
```

What would you have added if you wanted the new data set to have players ordered by overall score?

Coding Style, Commenting, Assigning

Sometimes, we just want to print a short table to the screen, and don't need to save it as a new dataset. For example, if we want to see the top 5 players in terms of Dribbling rating, we could do this:

```
Fifadata %>%
  select(Name, Dribbling) %>%
  arrange(desc(Dribbling)) %>%
  head(5)
```

```
## # A tibble: 5 x 2
##   Name       Dribbling
##   <chr>          <dbl>
## 1 L. Messi          97
## 2 Neymar Jr         96
## 3 E. Hazard         95
## 4 Isco              94
## 5 Y. Brahimi        93
```

We could also save this as an object and to print later, and can jazz it up with kable().

```
dribtable <- Fifadata %>%
  select(Name, Dribbling) %>%
  arrange(desc(Dribbling)) %>%
  head(5)

dribtable %>% kable(booktabs = TRUE)
```

Name	Dribbling
L. Messi	97
Neymar Jr	96
E. Hazard	95
Isco	94
Y. Brahimi	93

This is a short set of commands and may not be too difficult to parse. However, we should get in the habit of providing documentation for our code - short comments about what is going on in a chunk. If you look at the calendar query project code for example, the documentation helps you understand what is going on with a lot of new commands.

Here, this could be as easy as:

```
# Obtain table of top 5 Dribblers
dribtable <- Fifadata %>%
  select(Name, Dribbling) %>%
  arrange(desc(Dribbling)) %>%
  head(5)

#Print table
dribtable %>% kable(booktabs = TRUE)
```

Name	Dribbling
L. Messi	97
Neymar Jr	96
E. Hazard	95
Isco	94
Y. Brahimi	93

In addition, we want to be sure our code follows the general coding style we are using for class. I.E. we don't want our code to look like this:

```
# Obtain table of top 5 Dribblers and print
dribtable <- Fifadata %>% select(Name, Dribbling) %>% arrange(desc(Dribbling)) %>% head(5) %>% kable(bo
```

This runs off the page. Even if this was split between two lines (as below), it's still easier to read in the format with just ONE pipe operator per line. Again, the class coding style has generally one %>% or + per line.

```
# Obtain table of top 5 Dribblers and print
dribtable <- Fifadata %>% select(Name, Dribbling) %>% arrange(desc(Dribbling)) %>%
  head(5) %>% kable(booktabs = TRUE)
```

Various R style guides exist, as well as style guides for other programming languages. You can check out some listed below. Our class is using the first one - the tidyverse style guide. The pipes sub-page in particular might be of interest as you figure out what structure to use.

- Tidyverse Style guide
- Google R Style guide
- Google Python Style guide

Back to Wrangling Examples

What happens in the following code chunk? What comment could you write for the chunk to describe it?

```
# What goes here?
Fifadata %>%
  group_by(PreferredFoot) %>%
  na.omit() %>%
  summarize(meanAge = mean(Age), Count = n())
```

```
## # A tibble: 2 x 3
##   PreferredFoot meanAge Count
##   <chr>           <dbl> <int>
## 1 Left             25.1  4162
## 2 Right            25.1 13756
```

And for this example?

```
# What goes here?
Fifadata %>%
  skim(Overall)
```

Variable type: numeric

var	n	na	mean	sd	p0	p25	p50	p75	p100
Overall	18207	0	66.24	6.91	46	62	66	71	94

What about here?

```
# What goes here?
Fifadata %>%
  filter(Position == "GK") %>%
  select(Name,
         GKDiving,
         GKHandling,
         GKKicking,
         GKPositioning,
         GKReflexes) %>%
  mutate(GKSkillAvg = (GKDiving + GKHandling + GKKicking +
                         GKPositioning + GKReflexes)/5) %>%
  arrange(desc(GKSkillAvg)) %>%
  head(10)
```

```
## # A tibble: 10 x 7
##    Name        GKDiving GKHandling GKKicking GKPositioning GKReflexes GKSkillAvg
##    <chr>          <dbl>      <dbl>     <dbl>          <dbl>      <dbl>      <dbl>
##  1 De Gea            90         85        87             88         94       88.8
##  2 M. Neuer          90         86        91             87         87       88.2
##  3 M. ter Ste~       87         85        88             85         90       87
##  4 J. Oblak          86         92        78             88         89       86.6
##  5 Ederson           85         80        91             82         87       85
##  6 T. Courtois       85         91        72             86         88       84.4
##  7 G. Buffon         88         87        74             90         83       84.4
##  8 Alisson           83         81        85             84         88       84.2
##  9 S. Handano~       87         86        69             89         89       84
## 10 K. Navas          90         81        75             82         90       83.6
```

How would you look at the worst 8 goal keepers?

What would need to be different if you wanted to do the *select* before the *filter* to get the same result?

What could be done differently if you didn't want to keep all the original GK variables, just name and the new average?

What would you add to make the table `nicer` in your pdf?

Fixing position

```
mosaic::tally(~ Position, data = Fifadata)
```

```
## Registered S3 method overwritten by 'mosaic':
##   method                           from
##   fortify.SpatialPolygonsDataFrame ggplot2
```

```
## Position
##  CAM   CB  CDM   CF   CM   GK  LAM   LB  LCB  LCM  LDM   LF   LM   LS   LW  LWB
##  958 1778  948   74 1394 2025   21 1322  648  395  243   15 1095  207  381   78
##  RAM   RB  RCB  RCM  RDM   RF   RM   RS   RW  RWB   ST <NA>
##   21 1291  662  391  248   16 1124  203  370   87 2152   60
```

We can see that there are many positions in the overall data set - including some missing values (which we didn't see in the `small` version of the data set). How can we create just four categories (and remove missing values) from these values? We want to create forwards, midfielders, defenders, and goalkeepers.

From a website about the game, I found the following suggested breakdown:

- FWD - CF, LF, LS, LW, RF, RS, RW, ST
- MID - CAM, CDM, CM, LAM, LCM, LDM, LM, RAM, RCM, RDM, RM
- DEF - CB, LB, LCB, LWB, RB, RCB, RWB
- GK - GK

We also have some NA values to deal with. However, since those players won't be included in any plots using Position, we can also just remove them from the data set we are creating.

We will create a new position variable using this breakdown with the case_when() expression. Since MID has the most options, I will leave that as the final option (meaning all other cases get assigned that).

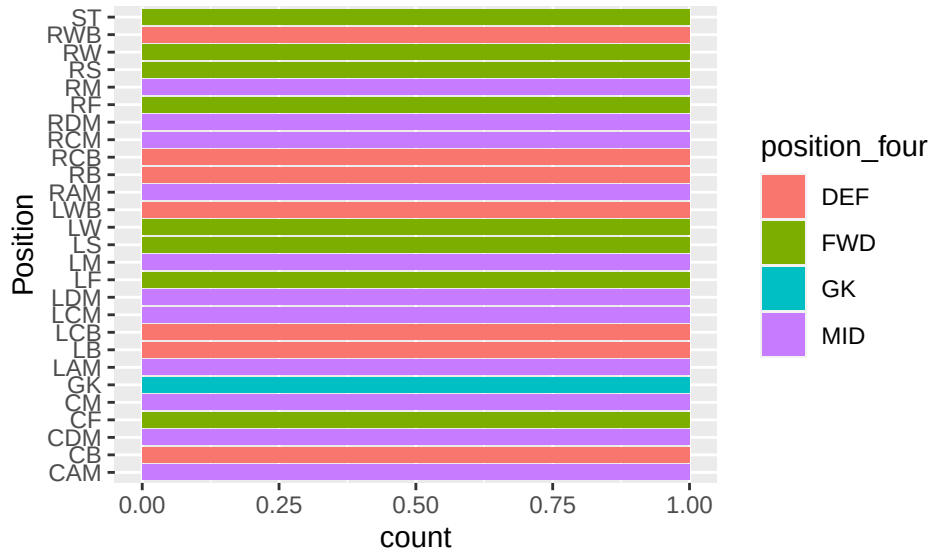
```
# Add new position variable
# Remove position NAs
Fifadata <- Fifadata %>%
  filter(Position != "NA") %>%
  mutate(position_four = case_when(
    Position == "GK" ~ "GK",
    Position %in% c("CF","LF","LS","LW","RF","RS","RW","ST") ~ "FWD",
    Position %in% c("CB","LB","LCB","LWB","RB","RCB","RWB") ~ "DEF",
    TRUE ~ "MID"
    )
  )

# Check new position breakdown
Fifadata %>%
  group_by(position_four) %>%
  summarize(count = n())
```

```
## # A tibble: 4 x 2
##   position_four count
##   <chr>         <int>
## 1 DEF            5866
## 2 FWD            3418
## 3 GK             2025
## 4 MID            6838
```

Note: I could have called the new data set by a new name, and I could have overwritten the position variable, but I chose not to for several reasons. First, I was just removing 60 observations that I won't need going forward, and adding a new variable, not dropping a large number. Often, it will be prudent to save a new, smaller data set with a new name. You need to be careful overwriting variables to make sure you get what you expect. After making sure the code works, you can always redo it to overwrite once you are sure it works.

How can I check that it worked? I need to be sure all the positions went where I expected. There are many ways to check. Here is a way to check with a plot.

```
ggplot(data = Fifadata, aes(x = Position)) +
  geom_bar(aes(fill = position_four), position = "fill") +
  coord_flip()
```



I could make a table too.

```
Fifadata %>%
  group_by(position_four, Position) %>%
  summarize(count = n())
```

```
## `summarise()` has grouped output by 'position_four'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 27 x 3
## # Groups:   position_four [4]
##    position_four Position count
##    <chr>         <chr>    <int>
##  1 DEF           CB        1778
##  2 DEF           LB        1322
##  3 DEF           LCB        648
##  4 DEF           LWB         78
##  5 DEF           RB        1291
##  6 DEF           RCB        662
##  7 DEF           RWB         87
##  8 FWD           CF          74
##  9 FWD           LF          15
## 10 FWD           LS         207
## # ... with 17 more rows
```

```
#2-way table might be useful as well
mosaic::tally(Position ~ position_four, data = Fifadata)
```

```
##          position_four
## Position  DEF  FWD   GK  MID
##      CAM    0    0    0  958
##      CB  1778    0    0    0
##      CDM    0    0    0  948
##      CF     0   74    0    0
##      CM     0    0    0 1394
##      GK     0    0 2025    0
##      LAM    0    0    0   21
```

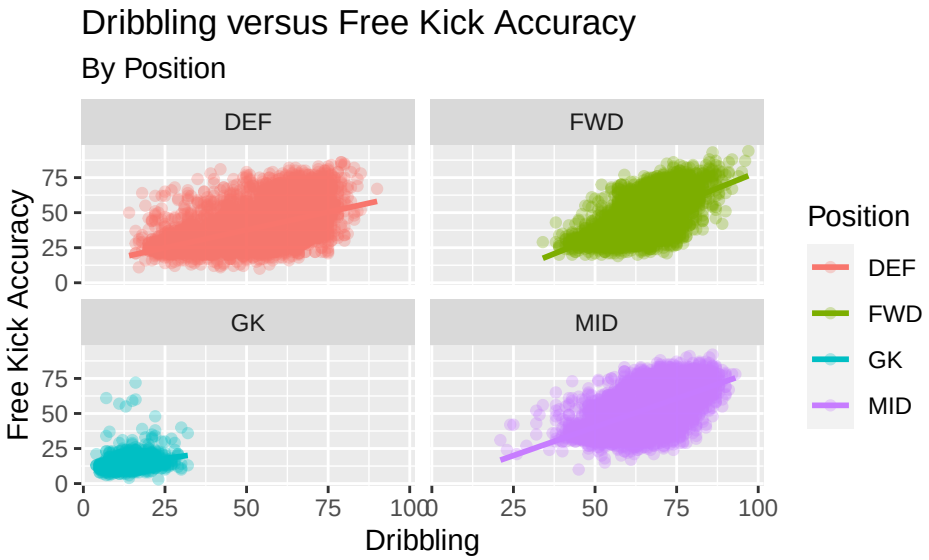
```
##       LB  1322    0    0    0
##       LCB  648    0    0    0
##       LCM    0    0    0  395
##       LDM    0    0    0  243
##       LF     0   15    0    0
##       LM     0    0    0 1095
##       LS     0  207    0    0
##       LW     0  381    0    0
##       LWB   78    0    0    0
##       RAM    0    0    0   21
##       RB  1291    0    0    0
##       RCB  662    0    0    0
##       RCM    0    0    0  391
##       RDM    0    0    0  248
##       RF     0   16    0    0
##       RM     0    0    0 1124
##       RS     0  203    0    0
##       RW     0  370    0    0
##       RWB   87    0    0    0
##       ST     0 2152    0    0
```

Now we can go back to the original plot with position I was interested in last week.

With so many data points, I also adjusted the point transparency. Darker points mean more values are present. I used facet_wrap because it was hard to see different colors when points were on top of each other.

```
ggplot(data = Fifadata, aes(x = Dribbling, y = FKAccuracy, color = position_four)) +
  geom_point(alpha = 0.3) +
  geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~ position_four) +
  labs(title = "Dribbling versus Free Kick Accuracy",
       subtitle = "By Position",
       y = "Free Kick Accuracy",
       color = "Position")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Ok. Let's briefly look at some date examples with lubridate, before we finish with some join and pivot examples.

Storms with Dates

You saw the nasaweather package for homework. The storms data set there is a filtered version of a storms data set in the *dplyr* package, which is loaded when we load the tidyverse package. We'll use this version since it has more data. The storms data set contains dates but not in a "date" format.

```
data(storms)
names(storms)
```

```
##  [1] "name"                          "year"
##  [3] "month"                         "day"
##  [5] "hour"                          "lat"
##  [7] "long"                          "status"
##  [9] "category"                      "wind"
## [11] "pressure"                      "tropicalstorm_force_diameter"
## [13] "hurricane_force_diameter"
```

```
storms <- storms %>% mutate(date = ymd(paste(year,month,day)))
glimpse(storms)
```

```
## Rows: 11,859
## Columns: 14
## $ name                          <chr> "Amy", "Amy", "Amy", "Amy", "Amy", "Amy",~
## $ year                          <dbl> 1975, 1975, 1975, 1975, 1975, 1975, 1975,~
## $ month                         <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,~
## $ day                           <int> 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 2~
## $ hour                          <dbl> 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18,~
## $ lat                           <dbl> 27.5, 28.5, 29.5, 30.5, 31.5, 32.4, 33.3,~
## $ long                          <dbl> -79.0, -79.0, -79.0, -79.0, -78.8, -78.7,~
## $ status                        <chr> "tropical depression", "tropical depressi~
## $ category                      <ord> -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, ~
## $ wind                          <int> 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 4~
## $ pressure                      <int> 1013, 1013, 1013, 1013, 1012, 1012, 1011,~
## $ tropicalstorm_force_diameter <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
## $ hurricane_force_diameter     <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
## $ date                          <date> 1975-06-27, 1975-06-27, 1975-06-27, 1975~
```

We can paste the date information together, and use the lubridate ymd() function to convert the variable to a date object. Note that I could have used dmy(), or whatever order I wanted, as long as it matched my paste.

Now that we have a date as a date object, we can do typical "date" computations - you can subtract dates to find time differences for example. What happens in the next chunk?

```
storms2 <- storms %>%
  # including year in the group_by is important...why?
  group_by(name, year) %>%
  summarize(startdate = min(date), enddate = max(date),
            length = (enddate - startdate)/ddays())
```

```
## `summarise()` has grouped output by 'name'. You can override using the
## `.groups` argument.
```

```
head(storms2, 10)
```

```
## # A tibble: 10 x 5
## # Groups:   name [10]
##    name      year startdate  enddate    length
##    <chr>    <dbl> <date>     <date>      <dbl>
##  1 AL011993  1993 1993-05-31 1993-06-02      2
##  2 AL012000  2000 2000-06-07 2000-06-08      1
##  3 AL021992  1992 1992-06-25 1992-06-26      1
##  4 AL021994  1994 1994-07-20 1994-07-21      1
##  5 AL021999  1999 1999-07-02 1999-07-03      1
##  6 AL022000  2000 2000-06-23 2000-06-25      2
##  7 AL022001  2001 2001-07-11 2001-07-12      1
##  8 AL022003  2003 2003-06-11 2003-06-11      0
##  9 AL022006  2006 2006-07-17 2006-07-18      1
## 10 AL031987  1987 1987-08-09 1987-08-17      8
```

```
storms2 %>% filter(name == "Ana")
```

```
## # A tibble: 7 x 5
## # Groups:   name [1]
##   name   year startdate  enddate    length
##   <chr> <dbl> <date>     <date>      <dbl>
## 1 Ana    1979 1979-06-19 1979-06-24      5
## 2 Ana    1985 1985-07-15 1985-07-19      4
## 3 Ana    1991 1991-07-02 1991-07-05      3
## 4 Ana    1997 1997-06-30 1997-07-04      4
## 5 Ana    2003 2003-04-21 2003-04-24      3
## 6 Ana    2009 2009-08-11 2009-08-16      5
## 7 Ana    2015 2015-05-09 2015-05-11      2
```

What about here?

```
storms3 <- storms2 %>% arrange(enddate)
head(storms3, 10)
```

```
## # A tibble: 10 x 5
## # Groups:   name [10]
##    name      year startdate  enddate    length
##    <chr>    <dbl> <date>     <date>      <dbl>
##  1 Amy       1975 1975-06-27 1975-07-04      7
##  2 Caroline  1975 1975-08-24 1975-09-01      8
##  3 Doris     1975 1975-08-29 1975-09-04      6
##  4 Belle     1976 1976-08-06 1976-08-10      4
##  5 Gloria    1976 1976-09-26 1976-10-04      8
##  6 Anita     1977 1977-08-29 1977-09-03      5
##  7 Clara     1977 1977-09-05 1977-09-11      6
##  8 Evelyn    1977 1977-10-13 1977-10-15      2
##  9 Amelia    1978 1978-07-30 1978-08-01      2
## 10 Bess      1978 1978-08-05 1978-08-08      3
```

And finally, here?

```
storms4 <- storms2 %>% arrange(desc(length))
head(storms4, 10)
```

```
## # A tibble: 10 x 5
## # Groups:   name [10]
##    name      year startdate  enddate    length
```

```
##    <chr>    <dbl> <date>     <date>      <dbl>
##  1 Nadine    2012 2012-09-10 2012-10-03     23
##  2 Ivan      2004 2004-09-02 2004-09-24     22
##  3 Alberto   2000 2000-08-03 2000-08-23     20
##  4 Kyle      2002 2002-09-23 2002-10-12     19
##  5 Frederic  1979 1979-08-29 1979-09-14     16
##  6 Georges   1998 1998-09-15 1998-10-01     16
##  7 Lee       2017 2017-09-14 2017-09-30     16
##  8 Edouard   1996 1996-08-19 1996-09-03     15
##  9 Emily     1993 1993-08-22 1993-09-06     15
## 10 Harvey    2017 2017-08-17 2017-09-01     15
```

Joins

Joins are ways of putting two or more data sets together. They are very useful for allowing the incorporation of data from different sources, but also to allow us to store data more effectively.

For the un_votes data set that we saw previously, for example, there were 2 data sets that basically provided supplementary information to a primary data set. While the primary data set had a row for every country-vote pair, the other data sets had information about each vote. Such information would be repeated (duplicated) for every country-vote pair where the vote was constant. Thus, it's more effective to store this information separately - provided you know how to recombine the data sets when you need the information.

Keys are important for linking datasets together. These are variables shared in common between the data sets that allow for reference between them.

The Fifadata set above was loaded in from a website where I am hosting the data. This is data from 2019. More recent data exists from 2021 (downloaded from Kaggle). Can we combine the files?

```
Fifadata19 <- Fifadata
Fifadata21 <- read_csv("data/fifa21data.csv")
```

```
## Rows: 17125 Columns: 107
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (52): Name, Nationality, Club, BP, Position, Player Photo, Club Logo, Fl...
## dbl (55): ID, Age, OVA, BOV, POT, Growth, Attacking, Crossing, Finishing, He...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
dim(Fifadata19)
```

```
## [1] 18147    41
```

```
dim(Fifadata21)
```

```
## [1] 17125   107
```

The two data sets have different numbers of observations and variables. They do share a "Name" column. Since the time periods are different, we can't combine based on Age, Weight, etc. and even things like "Clubs" may have changed. Let's see how we do trying to join them based on Name, after selecting just a few variables out of each dataset.

```
Fifadata21 <- Fifadata21 %>%
  select(Name, Age, OVA, Dribbling, Agility)
```

```
Fifadata19 <- Fifadata19 %>%
  select(Name, Age, Overall, Dribbling, Agility)
```

Now, I've kept some of the *same* variables in each data set so we can compare over time. However, I don't want to join based on those variables. We'll join just using Name. That's why the *by* argument is so important. Don't just let the computer pick columns. Set it yourself.

```
FifaCombined <- Fifadata19 %>%
  inner_join(Fifadata21, by = "Name")

glimpse(FifaCombined)
```

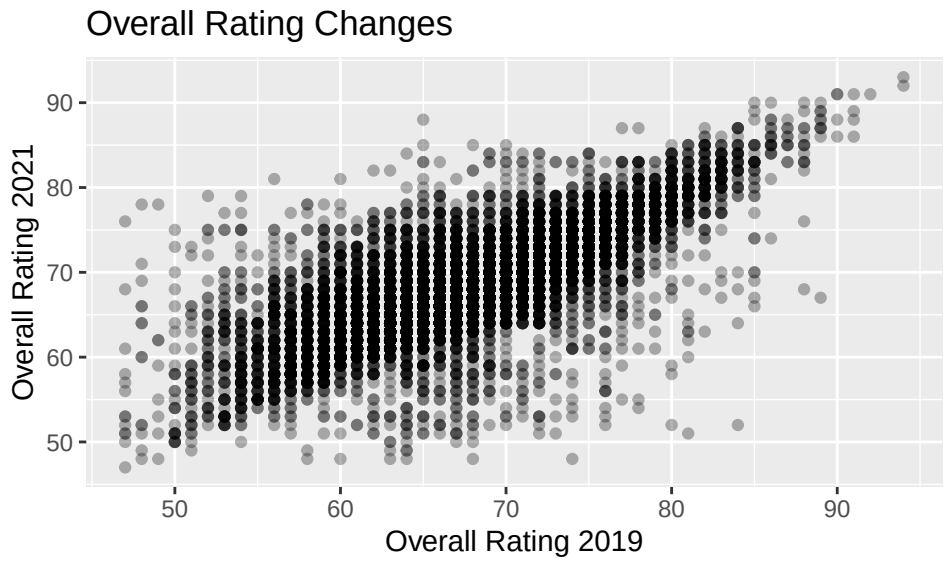
```
## Rows: 10,340
## Columns: 9
## $ Name        <chr> "L. Messi", "Cristiano Ronaldo", "Neymar Jr", "De Gea", "K~
## $ Age.x       <dbl> 31, 33, 26, 27, 27, 27, 32, 25, 29, 28, 32, 24, 24, 24, 27~
## $ Overall     <dbl> 94, 94, 92, 91, 91, 91, 91, 90, 90, 90, 90, 89, 89, 89, 89~
## $ Dribbling.x <dbl> 97, 88, 96, 18, 86, 95, 63, 12, 85, 81, 89, 92, 80, 80, 88~
## $ Agility.x   <dbl> 91, 87, 96, 60, 79, 95, 78, 67, 78, 70, 92, 91, 71, 71, 90~
## $ Age.y       <dbl> 33, 35, 28, 29, 29, 29, 34, 27, 31, 30, 34, 26, 26, 20, 29~
## $ OVA         <dbl> 93, 92, 91, 86, 91, 88, 89, 91, 91, 88, 86, 88, 88, 67, 87~
## $ Dribbling.y <dbl> 96, 88, 95, 18, 88, 93, 65, 12, 85, 80, 86, 91, 80, 62, 87~
## $ Agility.y   <dbl> 91, 87, 96, 63, 78, 92, 78, 67, 77, 63, 85, 92, 69, 67, 91~
```

Note how the variables have .x or .y added to them depending on which dataset they came from. We can adjust with rename.

```
FifaCombined <- FifaCombined %>%
  rename(Age19 = Age.x,
         OVA19 = Overall,
         Dribbling19 = Dribbling.x,
         Agility19 = Agility.x,
         Age21 = Age.y,
         OVA21 = OVA,
         Dribbling21 = Dribbling.y,
         Agility21 = Agility.y)
```

Now we can consider differences, if any, between the scores in the game on the attributes here and see what has changed, if anything. For example,

```
ggplot(FifaCombined, aes(x = OVA19, y = OVA21)) +
  geom_point(alpha = 0.3) +
  labs(x = "Overall Rating 2019",
       y = "Overall Rating 2021",
       title = "Overall Rating Changes")
```



Or maybe we just want to compute the difference and see some statistics about it. We'll do 2021 rating minus 2019 rating so that positive numbers show an improvement between the two years (at least in how the game was rating folks).

```
FifaCombined %>%
  mutate(OverallChange = OVA21 - OVA19) %>%
  skim(OverallChange)
```

Variable type: numeric

var	n	na	mean	sd	p0	p25	p50	p75	p100
OverallChange	10340	0	0.62	4.77	-32	-2	0	3	30

The agility and dribbling variables are included here so you have some options to play around with if you are looking through the notes.

When we first loaded the Fifadata21 data set, some of the variable names included spaces and other symbols. That's not ideal. R can protest and it becomes difficult to work with the data. How could we fix this at the outset? Try janitor's clean_names() function. Let's reload the data (since we overwrote it with select) and see what this does.

```
Fifadata21 <- read_csv("data/fifa21data.csv")
```

```
## Rows: 17125 Columns: 107
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (52): Name, Nationality, Club, BP, Position, Player Photo, Club Logo, Fl...
## dbl (55): ID, Age, OVA, BOV, POT, Growth, Attacking, Crossing, Finishing, He...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
names(Fifadata21)
```

```
##   [1] "ID"               "Name"             "Age"
##   [4] "OVA"              "Nationality"      "Club"
##   [7] "BOV"              "BP"               "Position"
##  [10] "Player Photo"     "Club Logo"        "Flag Photo"
##  [13] "POT"              "Team & Contract"  "Height"
```

```
##  [16] "Weight"           "foot"             "Growth"
##  [19] "Joined"           "Loan Date End"    "Value"
##  [22] "Wage"             "Release Clause"   "Contract"
##  [25] "Attacking"        "Crossing"         "Finishing"
##  [28] "Heading Accuracy" "Short Passing"    "Volleys"
##  [31] "Skill"            "Dribbling"        "Curve"
##  [34] "FK Accuracy"      "Long Passing"     "Ball Control"
##  [37] "Movement"         "Acceleration"     "Sprint Speed"
##  [40] "Agility"          "Reactions"        "Balance"
##  [43] "Power"            "Shot Power"       "Jumping"
##  [46] "Stamina"          "Strength"         "Long Shots"
##  [49] "Mentality"        "Aggression"       "Interceptions"
##  [52] "Positioning"      "Vision"           "Penalties"
##  [55] "Composure"        "Defending"        "Marking"
##  [58] "Standing Tackle"  "Sliding Tackle"   "Goalkeeping"
##  [61] "GK Diving"        "GK Handling"      "GK Kicking"
##  [64] "GK Positioning"   "GK Reflexes"      "Total Stats"
##  [67] "Base Stats"       "W/F"              "SM"
##  [70] "A/W"              "D/W"              "IR"
##  [73] "PAC"              "SHO"              "PAS"
##  [76] "DRI"              "DEF"              "PHY"
##  [79] "Hits"             "LS"               "ST"
##  [82] "RS"               "LW"               "LF"
##  [85] "CF"               "RF"               "RW"
##  [88] "LAM"              "CAM"              "RAM"
##  [91] "LM"               "LCM"              "CM"
##  [94] "RCM"              "RM"               "LWB"
##  [97] "LDM"              "CDM"              "RDM"
## [100] "RWB"              "LB"               "LCB"
## [103] "CB"               "RCB"              "RB"
## [106] "GK"               "Gender"
```

```
Fifadata21 <- Fifadata21 %>%
  janitor::clean_names()
names(Fifadata21)
```

```
##   [1] "id"               "name"             "age"
##   [4] "ova"              "nationality"      "club"
##   [7] "bov"              "bp"               "position"
##  [10] "player_photo"     "club_logo"        "flag_photo"
##  [13] "pot"              "team_contract"    "height"
##  [16] "weight"           "foot"             "growth"
##  [19] "joined"           "loan_date_end"    "value"
##  [22] "wage"             "release_clause"   "contract"
##  [25] "attacking"        "crossing"         "finishing"
##  [28] "heading_accuracy" "short_passing"    "volleys"
##  [31] "skill"            "dribbling"        "curve"
##  [34] "fk_accuracy"      "long_passing"     "ball_control"
##  [37] "movement"         "acceleration"     "sprint_speed"
##  [40] "agility"          "reactions"        "balance"
##  [43] "power"            "shot_power"       "jumping"
##  [46] "stamina"          "strength"         "long_shots"
##  [49] "mentality"        "aggression"       "interceptions"
##  [52] "positioning"      "vision"           "penalties"
##  [55] "composure"        "defending"        "marking"
```

```
##  [58] "standing_tackle"  "sliding_tackle"   "goalkeeping"
##  [61] "gk_diving"        "gk_handling"      "gk_kicking"
##  [64] "gk_positioning"   "gk_reflexes"      "total_stats"
##  [67] "base_stats"       "w_f"              "sm"
##  [70] "a_w"              "d_w"              "ir"
##  [73] "pac"              "sho"              "pas"
##  [76] "dri"              "def"              "phy"
##  [79] "hits"             "ls"               "st"
##  [82] "rs"               "lw"               "lf"
##  [85] "cf"               "rf"               "rw"
##  [88] "lam"              "cam"              "ram"
##  [91] "lm"               "lcm"              "cm"
##  [94] "rcm"              "rm"               "lwb"
##  [97] "ldm"              "cdm"              "rdm"
## [100] "rwb"              "lb"               "lcb"
## [103] "cb"               "rcb"              "rb"
## [106] "gk"               "gender"
```

Everything goes to lowercase, and spaces are replaced by _. A few other problematic symbols are changed as well, such as A/W became a_w. If you have problematic variable names, do this EARLY in your wrangling process. Here, because the variables we were pulling did not have issues, I didn't run it.

Pivots / Reshaping Data

The pivot commands can be challenging to understand. The basic premise behind needing them is that sometimes the way in which we store data isn't quite the format we want for analysis. We need to be able to move between formats.

Your text describes narrow versus wide formats. We may often collect data in wide formats, but need to put it in narrow format for analysis. Or it might be in narrow format, but going back to wide helps you get summary statistics you want.

Pivot_longer and pivot_wider are the pivot functions in the tidyverse. Previous versions of the functions were called gather and spread. So if you see an example online about gather or spread, it's the same concepts.

For these examples, we'll use a really small data set - suppose these are self-esteem scores for 10 subjects evaluated at 3 different time points (t1, t2, t3).

```
data("selfesteem", package = "datarium")
selfesteem
```

```
## # A tibble: 10 x 4
##       id    t1    t2    t3
##    <int> <dbl> <dbl> <dbl>
##  1     1  4.01  5.18  7.11
##  2     2  2.56  6.91  6.31
##  3     3  3.24  4.44  9.78
##  4     4  3.42  4.71  8.35
##  5     5  2.87  3.91  6.46
##  6     6  2.05  5.34  6.65
##  7     7  3.53  5.58  6.84
##  8     8  3.18  4.37  7.82
##  9     9  3.51  4.40  8.47
## 10    10  3.04  4.49  8.58
```

This is a natural way to enter the data for data collection - there is one column for each time point. It's also a nice structure if we wanted to compute differences between time points. Or if we wanted to plot the

trajectory for a particular patient. This is the wide data format, even though it's only 4 columns. The time points are in separate columns, rather than having a tp column that specifies the time point and a value column for the esteem value.

However, if we wanted to do an analysis to assess differences in means between the time points (ANOVA), or to do anything where time point is considered a variable, we need the narrow format. Thus, we want to pivot_longer into that format. We should end up with 30 rows instead of 10, as each row becomes a subject-time point, not just a subject.

```
esteem_narrow <- selfesteem %>%
  pivot_longer(-id,
               names_to = "time_point",
               values_to = "esteem_score")
esteem_narrow
```

```
## # A tibble: 30 x 3
##       id time_point esteem_score
##    <int> <chr>             <dbl>
##  1     1 t1                 4.01
##  2     1 t2                 5.18
##  3     1 t3                 7.11
##  4     2 t1                 2.56
##  5     2 t2                 6.91
##  6     2 t3                 6.31
##  7     3 t1                 3.24
##  8     3 t2                 4.44
##  9     3 t3                 9.78
## 10     4 t1                 3.42
## # ... with 20 more rows
```

Variables that are NOT being "reshaped", i.e. that you just want to carry along, need to be denoted with the -, or you can instead identify columns to include with the cols argument. For example, the following code gives the same data set.

```
selfesteem %>%
  pivot_longer(cols = starts_with("t"),
               names_to = "time_point",
               values_to = "esteem_score")
```

```
## # A tibble: 30 x 3
##       id time_point esteem_score
##    <int> <chr>             <dbl>
##  1     1 t1                 4.01
##  2     1 t2                 5.18
##  3     1 t3                 7.11
##  4     2 t1                 2.56
##  5     2 t2                 6.91
##  6     2 t3                 6.31
##  7     3 t1                 3.24
##  8     3 t2                 4.44
##  9     3 t3                 9.78
## 10     4 t1                 3.42
## # ... with 20 more rows
```

```
# OR
selfesteem %>%
  pivot_longer(cols = t1:t3,
```

```
               names_to = "time_point",
               values_to = "esteem_score")
```

```
## # A tibble: 30 x 3
##       id time_point esteem_score
##    <int> <chr>             <dbl>
##  1     1 t1                 4.01
##  2     1 t2                 5.18
##  3     1 t3                 7.11
##  4     2 t1                 2.56
##  5     2 t2                 6.91
##  6     2 t3                 6.31
##  7     3 t1                 3.24
##  8     3 t2                 4.44
##  9     3 t3                 9.78
## 10     4 t1                 3.42
## # ... with 20 more rows
```

Choosing whether to use the - or specify the columns in some way to pivot depends on how many variables are involved in each.

If the data started in this format and we wanted to go from narrow to wide, we could use pivot_wider(). Here, you just have to identify the column that the split is occurring across and what variable values go with it.

```
esteem_wide <- esteem_narrow %>%
  pivot_wider(names_from = time_point,
              values_from = esteem_score)
esteem_wide
```

```
## # A tibble: 10 x 4
##       id    t1    t2    t3
##    <int> <dbl> <dbl> <dbl>
##  1     1  4.01  5.18  7.11
##  2     2  2.56  6.91  6.31
##  3     3  3.24  4.44  9.78
##  4     4  3.42  4.71  8.35
##  5     5  2.87  3.91  6.46
##  6     6  2.05  5.34  6.65
##  7     7  3.53  5.58  6.84
##  8     8  3.18  4.37  7.82
##  9     9  3.51  4.40  8.47
## 10    10  3.04  4.49  8.58
```

To illustrate with a closely related example, I'm going to add a variable here so we can think about different plots/summaries the formats could be used to generate. Suppose of the 10 subjects, the first 5 were under condition "A" and the last 5 were under condition "B".

```
selfesteem <- selfesteem %>%
  mutate(condition = c(rep(c("A","B"), each = 5)))

selfesteem
```

```
## # A tibble: 10 x 5
##       id    t1    t2    t3 condition
##    <int> <dbl> <dbl> <dbl> <chr>
##  1     1  4.01  5.18  7.11 A
```

```
##  2     2  2.56  6.91  6.31 A
##  3     3  3.24  4.44  9.78 A
##  4     4  3.42  4.71  8.35 A
##  5     5  2.87  3.91  6.46 A
##  6     6  2.05  5.34  6.65 B
##  7     7  3.53  5.58  6.84 B
##  8     8  3.18  4.37  7.82 B
##  9     9  3.51  4.40  8.47 B
## 10    10  3.04  4.49  8.58 B
```

With the data in this wide format, I could compute average esteem values for each individual across the time points and compare the means for condition A versus B. I could make boxplots from this to compare the conditions in terms of their average scores (a little silly since there are only 5 observations for each condition).

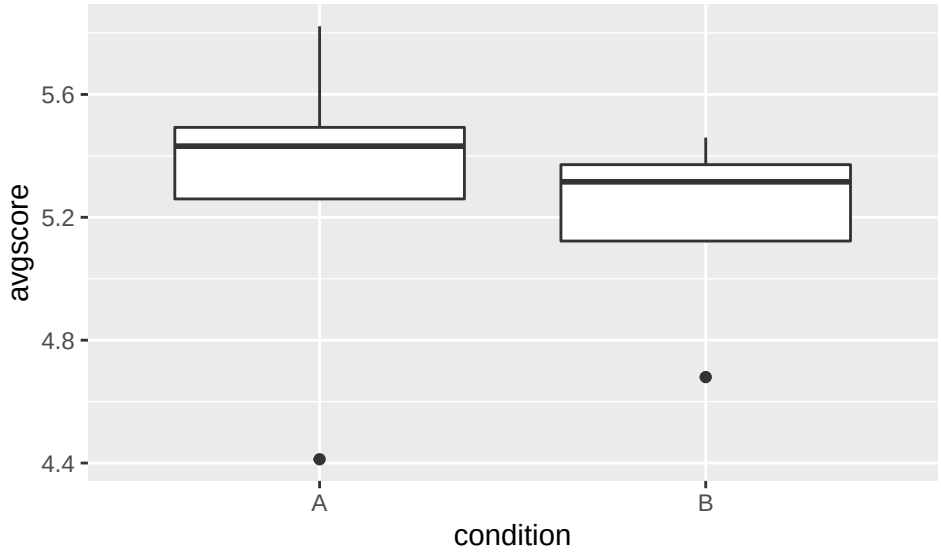
```
selfesteem %>%
  group_by(condition) %>%
  summarize(avgscore = (t1+t2+t3)/3)
```

```
## `summarise()` has grouped output by 'condition'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 10 x 2
## # Groups:   condition [2]
##    condition avgscore
##    <chr>        <dbl>
##  1 A             5.43
##  2 A             5.26
##  3 A             5.82
##  4 A             5.49
##  5 A             4.41
##  6 B             4.68
##  7 B             5.32
##  8 B             5.12
##  9 B             5.46
## 10 B             5.37
```

```
# To add on the plot
selfesteem %>%
  group_by(condition) %>%
  summarize(avgscore = (t1+t2+t3)/3) %>%
  ggplot(aes(x=condition, y = avgscore)) +
  geom_boxplot()
```

```
## `summarise()` has grouped output by 'condition'. You can override using the
## `.groups` argument.
```

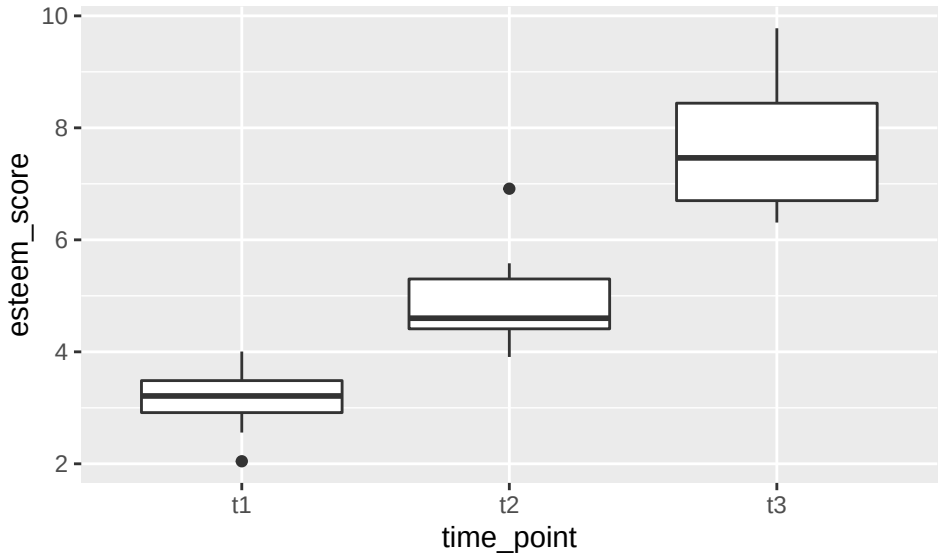


But anything I want to do to compare time points needs the data to be reshaped into the narrow format. This loses the repeated measures aspect of the data (i.e. be careful you don't interpret these as independent observations).

```
esteem_narrow <- selfesteem %>%
  pivot_longer(-c(id, condition), names_to = "time_point", values_to = "esteem_score")
esteem_narrow
```

```
## # A tibble: 30 x 4
##       id condition time_point esteem_score
##    <int> <chr>     <chr>              <dbl>
##  1     1 A         t1                  4.01
##  2     1 A         t2                  5.18
##  3     1 A         t3                  7.11
##  4     2 A         t1                  2.56
##  5     2 A         t2                  6.91
##  6     2 A         t3                  6.31
##  7     3 A         t1                  3.24
##  8     3 A         t2                  4.44
##  9     3 A         t3                  9.78
## 10     4 A         t1                  3.42
## # ... with 20 more rows
```

```
#To add the plot
esteem_narrow %>%
  ggplot(aes(x=time_point, y = esteem_score)) +
  geom_boxplot()
```



Here, I could plot all 15 esteem scores for each condition more easily. And I can get an overall average for condition much more easily than in the wide format.

```
esteem_narrow %>%
  group_by(condition) %>%
  summarize(average_esteem = mean(esteem_score))
```

```
## # A tibble: 2 x 2
##   condition average_esteem
##   <chr>              <dbl>
## 1 A                   5.28
## 2 B                   5.19
```

I could also get it by time point by condition.

```
esteem_narrow %>%
  group_by(condition, time_point) %>%
  summarize(average_esteem = mean(esteem_score))
```

```
## `summarise()` has grouped output by 'condition'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 6 x 3
## # Groups:   condition [2]
##   condition time_point average_esteem
##   <chr>     <chr>               <dbl>
## 1 A         t1                   3.22
## 2 A         t2                   5.03
## 3 A         t3                   7.60
## 4 B         t1                   3.06
## 5 B         t2                   4.84
## 6 B         t3                   7.67
```

For pivots, thinking about what your data needs to look like is key. Sketch out what the data set needs to look like on a sheet of paper or a chalk board. Once you get a sense of what the data needs to look like, you can start implementing steps to get it to that point.