Summary Sheet

AUTHOR
Julia Gallucci

Class 4

Factors and grouping:

We can mutate categorical columns to factors before grouping our data, to ensure that the levels of the factors are in the desired order. This allows for more intuitive and meaningful presentation of our data during summarization.

```
library(tidyverse)
CES_data %>%
  filter(cps19_province == "Ontario") %>%
  select(cps19_prov_gov_sat,
         cps19_prov_id,
         cps19_income_number) %>%
  mutate(cps19_prov_id = factor(cps19_prov_id,
         levels = c("Liberal", "Progressive Conservative","NDP","Green","Another party","
  group_by(cps19_prov_id) %>%
  summarise(median_income = median(cps19_income_number, na.rm = TRUE),
            count = n())
```

Here, we are mutating our provincial identification category from the CES data into a factor and providing specified levels that make logical sense.

Note: you can group data by more than one category (in the in-class example, we grouped by both provincial identification and provincial government satisfaction). Use `spread()` or `pivot_wider()` for improved readibility of summary table (see below)

```
CES_data %>%
  filter(cps19_province == "Ontario") %>%
  select(cps19_prov_gov_sat,
         cps19_prov_id,
         cps19_income_number) %>%
  mutate(cps19_prov_id = factor(cps19_prov_id, levels = c(
    "Liberal",
    "Progressive Conservative",
    "NDP",
    "Green",
    "Another party",
    "None",
    "Don't know/prefer not to answer"
  ))) %>%
  mutate(cps19_prov_gov_sat = factor(cps19_prov_gov_sat, levels = c(
```

```
    "Not at all satisfies",
    "Not very satisfied",
    "Fairly satisfied",
    "Very satisfied",
    "Don't know/prefer not to answer"
  ))) %>%
  group_by(cps19_prov_gov_sat, cps19_prov_id) %>%
  summarise(median_income = median(cps19_income_number, na.rm = TRUE)) %>%
  spread(key = cps19_prov_gov_sat,
         value = median_income)
```

Pivot

- Refers to the process of transforming data from a “long” format to a “wide” format or vice versa. This reshaping operation is used to restructure the data in a way that makes it easier to analyze or present.

pivot_longer:

`pivot_longer()` : used to transform data from a wide format to a long format. It “lengthens” data, increasing the number of rows and decreasing the number of columns. In the long format, each row represents a unique observation or data point, and different variables are stored in separate columns. This format is typically used when each row contains a single observation and there are multiple variables associated with each observation. The long format is often more suitable for data analysis and manipulation.

```
student_scores <-
  tibble(
  Student = c("Alice", "Julia", "Max", "Alice", "Julia", "Max"),
  Subject = c("Math", "Math", "Math", "Science", "Science", "Science"),
  Score = c(87, 77, 90, 85, 65, 95)
)
student_scores
```

```
# A tibble: 6 × 3
  Student Subject Score
  <chr>   <chr>   <dbl>
1 Alice   Math       87
2 Julia   Math       77
3 Max     Math       90
4 Alice   Science    85
5 Julia   Science    65
6 Max     Science    95
```

In this example, the long format has three columns: **`Student`**, **`Subject`**, and **`Score`**. Each row represents a student’s score in a particular subject.

Spread vs pivot_wider:

The `spread()` function works in the same way as `pivot_wider()`, widening the data, increasing the number of columns while decreasing the number of rows. This format is useful when summarizing or presenting data in a more compact and concise manner. Some differences:

spread(data, key, value)	**pivot_wider(data, names_from, values_from, <optional> values_fill)**
Converts a data frame from long to wide format based on key-value pairs.	Converts a data frame from long to wide format based on key-value pairs. It offers more flexibility and options compared to **`spread()`**, for ex. it allows for additional customization, such as specifying the default values for missing cells and handling duplicate values.
Arguments: • **`data`**: The input data frame. • **`key`**: The column that contains the keys used to create new columns. • **`value`**: The column that contains the values to be spread into the new columns.	Arguments: • **`data`**: The input data frame. • **`names_from`**: The column that contains the values to become the new column names. • **`values_from`**: The column that contains the values to be spread into the new columns. • **`values_fill`**: (optional) Specifies the default value for cells with missing values.

`pivot_wider()` is generally recommended for more advanced and flexible reshaping operations.

```
student_scores_wide <-
  student_scores %>%
  pivot_wider(names_from = Subject, values_from = Score)
student_scores_wide
```

```
# A tibble: 3 × 3
  Student  Math Science
  <chr>   <dbl>   <dbl>
1 Alice      87      85
2 Julia      77      65
3 Max        90      95
```

In this example, we specify **`names_from = Subject`** to indicate that the unique values in the "Subject" column will become the new column names. Similarly, we specify **`values_from = Score`** to indicate that

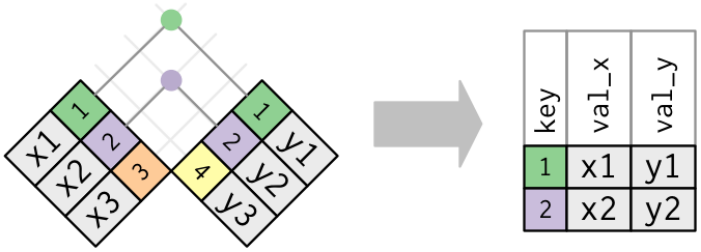
the values in the “Score” column will populate the new columns.

The resulting `student_scores_wide` tibble has three columns: “Student”, “Math”, and “Science”. Each row represents a student, and their scores in different subjects are displayed in the respective columns.

Join

used to combine or merge multiple data frames based on common variables or keys. There are several types of joins available, including inner join, left join, right join, and full join. The join operation brings together rows from different data frames that have matching values in the specified variables. **Keys** are the columns or variables **used to match and merge the data**

`inner_join()` returns **only the rows with matching values in both data frames**. It combines the rows from two data frames based on a common variable or key, excluding any rows that don’t have a match in the other data frame.

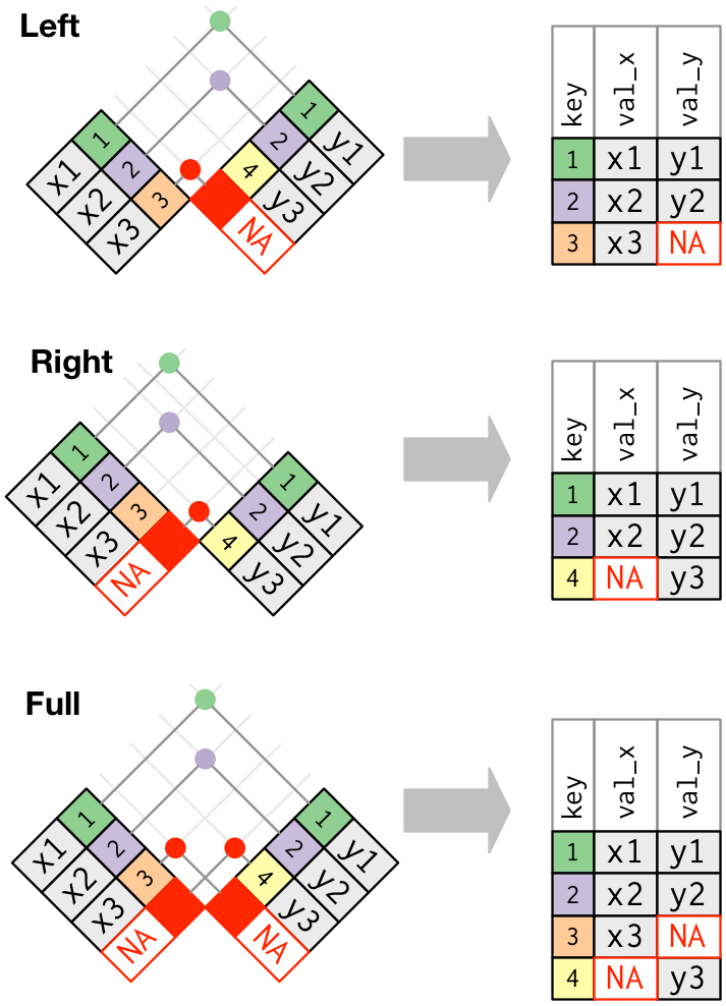


https://r4ds.had.co.nz/relational-data.html

`left_join()` returns **all rows from the left (or first) data fram**e and the matching rows from the right (or second) data frame based on a common variable or key. If there is **no match** for a row in the left data frame, the corresponding values from the **right data frame will be filled with NAs**.

`right_join()` returns **all rows from the right (or second) data frame** and the matching rows from the left (or first) data frame based on a common variable or key. If there is **no match** for a row in the right data frame, the corresponding values from the **left data frame will be filled with NAs.**

`full_join()` returns **all rows from both the left (or first) and right (or second) data frames**, combining them based on a common variable or key. If there is **no match for a row in either** data frame, the corresponding values from the other data frame will be **filled with NAs.**



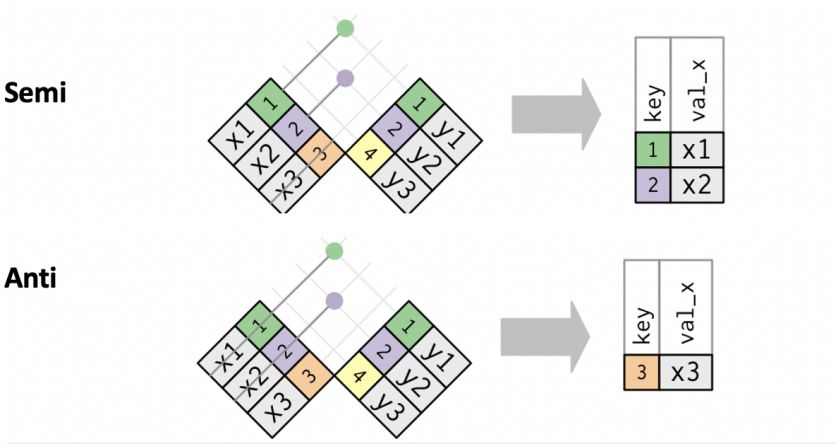
https://r4ds.had.co.nz/relational-data.html

Filter Joins

Allow you to filter one data frame based on matching values in another data frame. Filter joins are useful when you want to keep only the rows in one data frame that have matches in another data frame, effectively filtering the data based on the matching criteria.

`semi_join()` returns only the rows from the left (or first) data frame that have matching values in the right (or second) data frame.

`anti_join()` returns only the rows from the left (or first) data frame that do not have any matching values in the right (or second) data frame. It is essentially the *opposite of a regular join, as it keeps only the non-matching rows from the left data frame.*



https://r4ds.had.co.nz/relational-data.html

data.table

from data.table library, data tables are faster and more memory efficient than data.frames.

- `sapply(data_table, class)` can get info able class of data in your data table
- `haskey(data_table)` to check if your data table has a key
- `setkey(data_table, column_to_set)` to set a specified column in your table as a key
- `data_table[column_name with a certain condition]` to subset rows
- `data_table[, column_name]` to select columns
- `data_table[, list(column_name)]` to return column back as a data table

A **key** refers to one or more columns that are sorted in ascending order. The key helps in efficient indexing and sorting of the data. Here's an example:

```
dt <- data.table(ID = c(1, 2, 3),
                 Name = c("Alice", "Julia", "Max"),
                 Age = c(30, 25, 35))

# Set the key on the "ID" column
setkey(dt, ID)
dt
```

```
   ID  Name Age
1:  1 Alice  30
2:  2 Julia  25
3:  3   Max  35
```

After setting the key, the **data.table** is physically sorted by the "ID" column in ascending order. Once a key is set, it allows us to perform various operations that take advantage of the sorted order, leading to faster performance. For example, using the key, you can perform a **binary search** to efficiently locate rows based on the key column(s), as shown below.

```r
# Perform binary search using the key
search_id <- 3
result <- dt[ID == search_id]
result
```

```
   ID Name Age
1:  3  Max  35
```

Loading different types of data

- CSV `readr::read_csv()`
- XL `readxl::read_excel()`
- SPSS `haven::read_sav()`
- Stata `haven::read_dta()`
- SAS `haven::read_sas()`

can also give specifications, like skip lines using the `skip =` argument, change separator using the `sep =` argument, data does not have column names using the `col_names = FALSE` argument

to write data, use the following format:

```r
write.csv(variable_name, "what_you_want_to_save_your_output_file_as.csv")
```