

Session 3: Data Wrangling

R for Stata Users

Luiza Andrade, Rob Marty, Rony Rodriguez-Ramirez, Luis Eduardo San Martin, Leonardo Viotti

The World Bank – DIME | [WB Github](#)

May 2021



Table of contents

1. Introduction
2. Exploring your data
3. ID variables
4. Wrangling your data
5. Create variables
6. Appending and merging
7. Saving a dataframe
8. Factor variables
9. Reshaping

Introduction

Introduction

Goals of this session

- To organize data in a way that it will be easier to analyze it and communicate it.
- To get familiar with the packages bundled into the `tidyverse`.

Things to keep in mind

- We'll take you through the same steps we've taken when we were preparing the datasets used in this course.
- In most cases, your datasets won't be `tidy`.

Tidy data: A dataset is said to be tidy if it satisfies the following conditions:

1. Every column is a variable.
2. Every row is an observation.
3. Every cell is a single value.

Introduction

In this session, you'll be introduced to some basic concepts of data cleaning in R. We will cover:

1. Exploring a dataset;
2. Creating new variables;
3. Filtering and subsetting datasets;
4. Merging datasets;
5. Dealing with factor variables;
6. Saving data.

There are many other tasks that we usually perform as part of data cleaning that are beyond the scope of this session.

Introduction

Before we start, let's make sure we are all set:

1. Start a fresh RStudio session.
2. Open your RStudio project.

Notes for this session

- Most of our exercises for today focus on the tidyverse.
- The setup is a bit different than yesterday.
- Since we are "exploring" our data, we sometimes don't need to assign everything to an object.

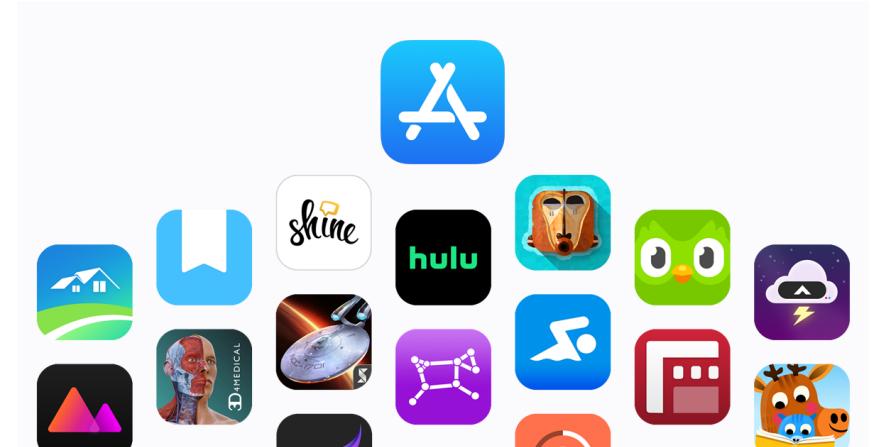
Introduction: Packages

Another important aspect to consider is R packages. Consider the following:

R is a new phone



R packages are apps in your phone



Introduction: Packages

To install a package you can run the following command:

```
# To Install  
install.packages("tidyverse")
```

- Unlike Stata, R packages need to be loaded in each R session that will use them.
- That means that, for example, a function that comes from the tidyverse cannot be used if the tidyverse meta-package has not been installed and loaded first.

To load a package you can run the following command:

```
# To Install  
library(tidyverse)
```

- Notice that we use double quotes for installing but not for loading a package.

Let's load the two following packages:

```
# If you haven't installed the packages uncomment the next line  
# install.packages("tidyverse")  
# install.packages("here")  
# install.packages("janitor")  
library(tidyverse)  
library(here)      # A package to work with relative file paths  
library(janitor)   # Additional data cleaning tools
```

Notes: Remember you should always load your packages before you start coding.

File paths

The `here` package allows you to interact with your working directory. It will look for the closest R Project and set its location as the working directory. That's why it is important to set your RStudio project correctly.

The goal of this package is to:

- Easily reference your files in project-oriented workflows.



Loading a dataset in R

Before we start wrangling our data, let's read our dataset. In R, we can use the `read.csv` function from Base R, or `read_csv` from the `readr` package if we want to load a CSV file. For this exercise, we are going to use the World Happiness Report (2015-2018)

Exercise 1: Load Data. Recap from yesterday's session.

Use either of the functions mentioned above and load the three WHR datasets from the `DataWork/DataSets/Raw/Un WHR` folder. Use the following notation for each dataset: `whrYY`.

- Remember to use `here()` to simplify the folder path.

How to do it?

```
whr15 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2015.csv"))
whr16 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2016.csv"))
whr17 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2017.csv"))
```

The pipe %>% operator

- "Piping" in R can be seen as "**chaining**." This means that we are invoking multiple method calls.
- Every time you have invoked a method (a function) this return an object that then is going to be used in the next pipe.

```
rony %>%
  wake_up(time = "5:30") %>%
  get_out_of_bed() %>%
  do_exercise() %>%
  shower() %>%
  get_dressed() %>%
  eat(meal = "breakfast", coffee = TRUE) %>%
  brush_teeth() %>%
  work(effort = "mininum")
```

```
work(
  brush_teeth(
    eat(
      get_dressed(
        shower(
          do_exercise(
            get_out_of_bed(
              wake_up(me, time = "5:30")
            ),
            )
          )
        ),
        meal = "breakfast", coffee = TRUE
      )
    ),
    effort = "mininum"
  )
```

The pip %>% operator

From R for Data Science by Wickham & Grolemund:

Pipes are a powerful tool for clearly expressing a sequence of multiple operations. The point of the pipe is to help you write code in a way that is easier to read and understand. [...] It focusses on verbs, not nouns. You can read this series of function compositions like it's a set of imperative actions.

The clean_names() function

The `clean_names()` function helps us big time when our variables names are pretty bad. For example, if we have a variable that is called `GDP_per_CApita_2015`, the `clean_names()` function will help us fix those messy names.

Tip: Use the `clean_names()` function in a pipe after you load a dataset as we did in the last slide. It will make sure column names are well-suited for use in R

```
whr15 <- whr15 %>%
  clean_names()
whr16 <- whr16 %>%
  clean_names()
whr17 <- whr17 %>%
  clean_names()
```

If we want to rename our variable manually, we could use:

```
whr15 <- whr15 %>%
  rename(
    var_newname = var_oldname
  )
```

Exploring your data

Exploring a data set

These are some useful functions from base R:

- `View()`: open the data set.
- `class()`: reports object type or type of data stored.
- `dim()`: reports the size of each one of an object's dimension.
- `names()`: returns the variable names of a dataset.
- `str()`: general information on an R object.
- `summary()`: summary information about the variables in a data frame.
- `head()`: shows the first few observations in the dataset.
- `tail()`: shows the last few observations in the dataset.

Some other useful functions from the tidyverse:

- `glimpse()`: get a glimpse of your data.

Load and show a dataset

We can just show our dataset using the name of the object; in this case, `whr15`.

```
whr15
```

```
## # A tibble: 158 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per-
##   <chr>    <chr>          <dbl>            <dbl>           <dbl>
## 1 Switze~ Weste~           1             7.59        0.0341       1.40
## 2 Iceland Weste~           2             7.56        0.0488       1.30
## 3 Denmark Weste~           3             7.53        0.0333       1.33
## 4 Norway  Weste~           4             7.52        0.0388       1.46
## 5 Canada   North~          5             7.43        0.0355       1.33
## 6 Finland Weste~           6             7.41        0.0314       1.29
## 7 Nether~ Weste~           7             7.38        0.0280       1.33
## 8 Sweden   Weste~           8             7.36        0.0316       1.33
## 9 New Ze~ Austr~           9             7.29        0.0337       1.25
## 10 Austra~ Austr~          10            7.28        0.0408       1.33
## # ... with 148 more rows, and 6 more variables: family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

Glimpse your data

Use `glimpse()` to get information about your variables (e.g., type, row, columns,)

```
whr15 %>%
  glimpse()
```

```
## Rows: 158
## Columns: 12
## $ country <chr> "Switzerland", "Iceland", "Denmark", "Norw~  
## $ region <chr> "Western Europe", "Western Europe", "Weste~  
## $ happiness_rank <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,~  
## $ happiness_score <dbl> 7.587, 7.561, 7.527, 7.522, 7.427, 7.406, ~  
## $ standard_error <dbl> 0.03411, 0.04884, 0.03328, 0.03880, 0.0355~  
## $ economy_gdp_per_capita <dbl> 1.39651, 1.30232, 1.32548, 1.45900, 1.3262~  
## $ family <dbl> 1.34951, 1.40223, 1.36058, 1.33095, 1.3226~  
## $ health_life_expectancy <dbl> 0.94143, 0.94784, 0.87464, 0.88521, 0.9056~  
## $ freedom <dbl> 0.66557, 0.62877, 0.64938, 0.66973, 0.6329~  
## $ trust_government_corruption <dbl> 0.41978, 0.14145, 0.48357, 0.36503, 0.3295~  
## $ generosity <dbl> 0.29678, 0.43630, 0.34139, 0.34699, 0.4581~  
## $ dystopia_residual <dbl> 2.51738, 2.70201, 2.49204, 2.46531, 2.4517~
```

Dimensions of your data

Dimensions of your data

Let's see first:

- **Dimensions of your data** (Rows and Columns):

```
dim(whr15)
```

```
## [1] 158 12
```

- **The number of distinct values of a particular variable:**

```
n_distinct(DATASET$variable, na.rm = TRUE)
```

The `$` sign is a subsetting operator. In R, we have three subsetting operators (`[[]`, `[`, and `$`). It is often used to access variables in a dataframe.

The `n_distinct` function allows us to count the number of unique values of a variable length of a vector. We included `na.rm = TRUE`, so we don't count missing values.

Dimensions of your data

Exercise 2: Identify distinct values of a variable in a dataset

Using the `n_distinct` function, can you tell how many unique values these variables of the whr15 dataset have?

1. Region
2. Country

How to do it?

```
n_distinct(whr15$country, na.rm = TRUE)
```

```
## [1] 158
```

```
n_distinct(whr15$region, na.rm = TRUE)
```

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

01:00

Dimensions of your data

We can also test whether the number of rows is equal to the number of distinct values in a specific variable as follows:

```
nrow(whr15)
```

```
## [1] 158
```

We can use the two functions (`nrow` and `n_distinct`) together to test if their result is the same.

```
n_distinct(whr15$country, na.rm = TRUE) == nrow(whr15)
```

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

```
n_distinct(whr16$country, na.rm = TRUE) == nrow(whr16)
```

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

```
n_distinct(whr17$country, na.rm = TRUE) == nrow(whr17)
```

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

Wrangling your data

dplyr::filter

- The `filter` function is used to subset rows in a dataset.

```
whr15 %>% filter(region == "Western Europe")
```

```
## # A tibble: 21 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per-
##   <chr>    <chr>          <dbl>            <dbl>        <dbl>             <dbl>
## 1 Switze~ Weste~           1            7.59       0.0341        1.40
## 2 Iceland Weste~          2            7.56       0.0488        1.30
## 3 Denmark Weste~          3            7.53       0.0333        1.33
## 4 Norway  Weste~          4            7.52       0.0388        1.46
## 5 Finland Weste~          6            7.41       0.0314        1.29
## 6 Nether~ Weste~          7            7.38       0.0280        1.33
## 7 Sweden  Weste~          8            7.36       0.0316        1.33
## 8 Austria Weste~          13           7.2        0.0375        1.34
## 9 Luxemb~ Weste~          17           6.95       0.0350        1.56
## 10 Ireland Weste~         18           6.94       0.0368        1.34
## # ... with 11 more rows, and 6 more variables: family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

dplyr::filter

Exercise 3: Filter the dataset.

- Use `filter()` to extract only rows in one of these regions: (1) Eastern Asia and (2) North America. Hint: use the `or` operator (`|`) to add conditions:

```
whr15 %>%
  filter(region == "Eastern Asia" | region == "North America")
```

A more elegant approach would be to use the `%in%` operator (equivalent to `inlist()` in Stata):

```
whr15 %>%
  filter(region %in% c("Eastern Asia", "North America"))
```

dplyr::filter missing cases

If case you want to remove (or identify) the missing cases for a specific variable, you can use `is.na()`.

- This function returns a value of true and false for each value in a data set.
- If the value is *NA* the `is.na()` function returns TRUE, otherwise, it returns FALSE.
- In this way, we can check NA values that can be used for other functions.
- We can also negate the function using `!is.na()` which indicates that we want to return those observations with no missing values in a variable.

The function syntax in a pipeline is as follows:

```
DATA %>%
  filter(is.na(VAR))
```

What are we returning here?

The observations that have missing values for the variable VAR.

dplyr::filter missing cases

Let's try filtering the whr15 data. Let's keep those observations that have information per region, i.e., no missing values.

```
whr15 %>%
  filter(!is.na(region)) %>%
  head(5)
```

```
## # A tibble: 5 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per~
##   <chr>    <chr>          <dbl>            <dbl>           <dbl>
## 1 Switzer~ Wester~             1            7.59        0.0341        1.40
## 2 Iceland   Wester~             2            7.56        0.0488        1.30
## 3 Denmark   Wester~             3            7.53        0.0333        1.33
## 4 Norway    Wester~             4            7.52        0.0388        1.46
## 5 Canada    North~              5            7.43        0.0355        1.33
## # ... with 6 more variables: family <dbl>, health_life_expectancy <dbl>,
## #   freedom <dbl>, trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

Notice that we are negating the function, i.e., `!is.na()`

In case we want to keep the observations that contains missing information we will only use `is.na()`.

Creating new variables

Creating new variables

In the tidyverse, we use the `mutate()` function. Let's say we want to have interactions:

```
whr15 %>%
  mutate(
    hap_hle = happiness_score * health_life_expectancy,
  )
```

This will add a new variable called `hap_hle` which is the interaction of happiness score and health life expectancy.

Creating new variables: Dummy variables

```
whr15 %>%
  mutate(happiness_score_6 = (happiness_score > 6))
```

What do you think is happening to this variable?

The variable we created contains either `TRUE` or `FALSE`.

If we wanted to have it as a numeric (1 or 0, respectively), we could include `as.numeric()`. However, the point of having logical variables is to treat them as numbers when relevant (for example as dummy variables in a regression) and as categories when relevant (for example in graphs)

```
whr15 %>%
  mutate(happiness_score_6 = as.numeric((happiness_score > 6)))
```

Finally, instead of using a random number such as 6, we can do the following:

```
whr15 %>%
  mutate(happiness_high_mean = as.numeric((happiness_score > mean(happiness_score))))
```

Some notes: mutate() vs transmute()

`mutate()` vs `transmute()`

Similar in nature but:

1. `mutate()` returns original and new columns (variables).
2. `transmute()` returns only the new columns (variables).

Creating variables by groups

In R, we can use `group_by()` before we mutate to group an estimation. So, for this example, we are going to pipe the following functions.

1. Group our data by the `region` variable.
2. Create a variable that would be the mean of `happiness_score` by each region.
3. Select the variables `country, region, happiness_score, mean_hap`.

Example With variables Output

```
DATASET %>%
  group_by(GROUPING VARIABLE) %>%
  mutate(
    NAME OF NEW VAR = mean(VARIABLE)
  ) %>%
  select(VAR1, VAR2, VAR3, VAR4)
```

Creating variables by groups

In R, we can use `group_by()` before we mutate to group an estimation. So, for this example, we are going to pipe the following functions.

1. Group our data by the `region` variable.
2. Create a variable that would be the mean of `happiness_score` by each region.
3. Select the variables `country, region, happiness_score, mean_hap`.

Example

With variables

Output

```
whr15 %>%
  group_by(region) %>%
  mutate(
    mean_hap = mean(happiness_score)
  ) %>%
  select(country, region, happiness_score, mean_hap)
```

Creating variables by groups

In R, we can use `group_by()` before we mutate to group an estimation. So, for this example, we are going to pipe the following functions.

1. Group our data by the `region` variable.
2. Create a variable that would be the mean of `happiness_score` by each region.
3. Select the variables `country`, `region`, `happiness_score`, `mean_hap`.

Example With variables Output

```
## # A tibble: 7 x 4
## # Groups:   region [2]
##   country     region   happiness_score  mean_hap
##   <chr>       <chr>           <dbl>      <dbl>
## 1 Switzerland Western Europe     7.59      6.69
## 2 Iceland     Western Europe     7.56      6.69
## 3 Denmark     Western Europe     7.53      6.69
## 4 Norway      Western Europe     7.52      6.69
## 5 Canada      North America    7.43      7.27
## 6 Finland     Western Europe     7.41      6.69
## 7 Netherlands Western Europe     7.38      6.69
```

Creating multiple variables at the same time

We can create multiple variables in an easy way. So, let's imagine that we want to estimate the mean value for the variables: happiness_score, health_life_expectancy, and trust_government_corruption.

How we can do it?

- We can use the function `across()`. It behaves this way: `across(VARS that you want to transform, FUNCTION to execute)`.
- `across()` should be always use inside `summarise()` or `mutate()`.

Across Output

```
vars <- c("happiness_score", "health_life_expectancy", "trust_government_corruption")  
  
wri15 %>%  
  group_by(region) %>%  
  summarise(  
    across(all_of(vars), mean)  
)
```

Creating multiple variables at the same time

We can create multiple variables in an easy way. So, let's imagine that we want to estimate the mean value for the variables: happiness_score, health_life_expectancy, and trust_government_corruption.

How we can do it?

- We can use the function `across()`. It behaves this way: `across(VARS that you want to transform, FUNCTION to execute)`.
- `across()` should be always use inside `summarise()` or `mutate()`.

Across Output

```
## # A tibble: 3 x 4
##   region           happiness_score health_life_expe~ trust_government~
##   <chr>                 <dbl>            <dbl>                  <dbl>
## 1 Australia and New Zealand     7.28            0.920                  0.393
## 2 Central and Eastern Europe    5.33            0.719                  0.0867
## 3 Eastern Asia                 5.63            0.877                  0.128
```

Creating variables: Exercise

00 : 45

Exercise 4: Create a variable called year that equals to the year of each dataframe .

- Use `mutate()`
- Remember to assign it to the same dataframe.

How to do it?

```
whr15 <- whr15 %>%
  mutate(
    year = 2015
  )

whr16 <- whr16 %>%
  mutate(
    year = 2016
  )

whr17 <- whr17 %>%
  mutate(
    year = 2017
  )
```

Appending and merging data sets

Appending and merging data sets

Now that we can identify the observations, we can combine the data set. Here are two functions to append objects by row

```
rbind(df1, df2, df3) # The base R function
```

```
bind_rows(df1, df2, df3) # The tidyverse function, making some improvements to base R
```

Exercise 5: Append data sets.

- Use the function `bind_rows` to append the three WHR datasets:

How to do it?

```
bind_rows(whr15, whr16, whr17)
```

What problems do you think we can have with these approach?

- One of the problems with binding rows like this is that, sometimes, some columns are not compatible.

00 : 45

36 / 58

Appending and merging data sets

Exercise 6: Fixing our variables and appending the data frames correctly.

Exercise 6a:

- Load the R data set `regions.RDS` from `DataWork/DataSets/Raw/Un WHR` using the `read_rds` function.

```
regions <- read_rds(here("DataWork", "DataSets", "Raw", "Un WHR", "regions.RDS"))
```

00:45

Appending and merging data sets

We can use the `left_join()` function to merge two dataframes. The function syntax is: `left_join(a_df, another_df, by = c("id_col1"))`.

A left join takes all the values from the first table, and looks for matches in the second table. If it finds a match, it adds the data from the second table; if not, it adds missing values. It is the equivalent of `merge, keep(master matched)` in Stata.

Exercise 6b:

- Now, we join the `regions` dataframe with the `whr17` dataframe.

```
whr17 <- whr17 %>%  
  left_join(regions, by = "country") %>%  
  select(country, region, everything())
```

Notes: Look at the `everything()` function. It takes all the variables from the dataframe and put them after country and region. In this way, `select` can be used to **order** columns!

Appending and merging data sets

Exercise 6c:

Check if there is any other country without region info:

- Only use pipes %>%
- And `filter()`
- Do not assign it to an object.

```
whr17 %>%
  filter(is.na(region))

## # A tibble: 2 x 14
##   country      region happiness_rank happiness_score whisker_high whisker_low economy_gdp_per_~
##   <chr>        <chr>          <dbl>            <dbl>           <dbl>           <dbl>           <dbl>
## 1 Taiwan Prov~ <NA>             33              6.42            6.49            6.35            1.43
## 2 Hong Kong S~ <NA>             71              5.47            5.55            5.39            1.55
## # ... with 7 more variables: family <dbl>, health_life_expectancy <dbl>, freedom <dbl>,
## #   generosity <dbl>, trust_government_corruption <dbl>, dystopia_residual <dbl>, year <dbl>
```

01:00

So we ended up with two countries with NAs

This is due to the name of the countries. The regions dataset doesn't have "Taiwan Province of China" nor "Hong Kong S.A.R., China" but "Taiwan" and "Hong Kong".

How do you think we should solve this?

- My approach would be to:
 1. fix the names of these countries in the whr17 dataset and;
 2. merge (left_join) it with the regions dataset.

Appendix: [case_when](#) and [mutate](#) for more information.

Appending and merging data sets

Finally, let's keep those relevant variables first and bind those rows.

Exercise 7: Bind all rows and create a panel called: `whr_panel`.

- Use `rbind()`
- Select the variables: `country`, `region`, `year`, `happiness_rank`, `happiness_score`, `economy_gdp_per_capita`, `health_life_expectancy`, `freedom` for each df, i.e., `whr15`, `whr16`, `whr17`.

```
keepvars <- c("country", "region", "year", "happiness_rank",
             "happiness_score", "economy_gdp_per_capita",
             "health_life_expectancy", "freedom")

whr15 <- select(whr15, all_of(keepvars))
whr16 <- select(whr16, all_of(keepvars))
whr17 <- select(whr17, all_of(keepvars))

whr_panel <- rbind(whr15, whr16, whr17)    # or bind_rows
```

01:00

Saving a dataset

Saving a dataset

- The data set you have now is the same data set we've been using for earlier sessions, so we can save it now
- As mentioned before, R data sets are often saved as csv.
- To save a dataset we can use the `write_csv` function from the tidyverse, or `write.csv` from base R.

The function takes the following structure:

```
write_csv(x, file, append = FALSE):
```

- `x`: the object (usually a data frame) you want to export to CSV
- `file`: the file path to where you want to save it, including the file name and the format (“.csv”)
- `append`: If FALSE, will create a new file. If TRUE, will append to an existing file.

Saving a dataset

Exercise 8: Save the dataset.

- Use `write_csv()`
- Use `here()`

```
# Save the whr data set

write_csv(
  whr_panel,
  here("DataWork", "DataSets", "Final", "whr_panel.csv")
)
```

- The problem with CSVs is that they cannot differentiate between `strings` and `factors`
- They also don't save factor orders
- Data attributes (which are beyond the scope of this training, but also useful to document data sets) are also lost.

Saving a dataset

The R equivalent of a `.dta` file is a `.rds` file. It can be saved and loaded using the following commands:

- `write_rds(object, file = "")`: Writes a single R object to a file.
- `read_rds(file)`: Load a single R object from a file.

```
# Save the data set

write_rds(
  whr_panel,
  here("DataWork", "DataSets", "Final", "whr_panel.Rds")
)
```

Thank you~

Appendix

Missing values in R

Quick Note:

- Missing values in R are treated differently than in Stata. They are represented by the NA symbol.
- Impossible values are represented by the symbol NaN which means 'not a number'.
- R uses the same symbol for character and numeric data.
- NA is not a string or a numeric value, but an indicator of missingness.
- NAs are contagious. This means that if you compare a number with NAs you will get NAs.
- Therefore, always remember the `na.rm = TRUE` argument if needed.

Other relevant functions: slice, subset, select

Arrange Slice Select Combining functions

Arrange: allows you to order by a specific column.

```
whr15 %>%
  arrange(region, country) %>%
  head(5)
```

```
## # A tibble: 5 x 8
##   country     region     year happiness_rank happiness_score economy_gdp_per_~
##   <chr>       <chr>      <dbl>          <dbl>            <dbl>
## 1 Australia  Australia ~ 2015           10        7.28        1.33
## 2 New Zealand Australia ~ 2015           9         7.29        1.25
## 3 Albania    Central an~ 2015          95        4.96        0.879
## 4 Armenia    Central an~ 2015          127        4.35        0.768
## 5 Azerbaijan Central an~ 2015          80         5.21        1.02
## # ... with 2 more variables: health_life_expectancy <dbl>, freedom <dbl>
```

Other relevant functions: slice, subset, select

Arrange Slice Select Combining functions

Slice: allows you to select, remove, and duplicate rows.

```
whr15 %>%
  slice(1:5) # to select the first 5 rows
```

```
## # A tibble: 5 x 8
##   country     region     year happiness_rank happiness_score economy_gdp_per_c~
##   <chr>       <chr>      <dbl>          <dbl>            <dbl>
## 1 Switzerland Western Europe 2015           1             7.59            1.40
## 2 Iceland     Western Europe 2015           2             7.56            1.30
## 3 Denmark     Western Europe 2015           3             7.53            1.33
## 4 Norway      Western Europe 2015           4             7.52            1.46
## 5 Canada      North America 2015           5             7.43            1.33
## # ... with 2 more variables: health_life_expectancy <dbl>, freedom <dbl>
```

You can also use `slice_head` and `slice_tail` to select the first or last rows respectively. Or `slice_sample` to randomly draw n rows.

Other relevant functions: slice, subset, select

Arrange Slice **Select** Combining functions

Select: allows you to select specific columns.

```
library(tidyverse)
  select(region, country, happiness_rank)
```

```
## # A tibble: 158 x 3
##   region           country  happiness_rank
##   <chr>            <chr>          <dbl>
## 1 Western Europe  Switzerland      1
## 2 Western Europe  Iceland         2
## 3 Western Europe  Denmark        3
## 4 Western Europe  Norway         4
## 5 North America   Canada         5
## 6 Western Europe  Finland        6
## 7 Western Europe  Netherlands    7
## 8 Western Europe  Sweden         8
## 9 Australia and New Zealand New Zealand     9
## 10 Australia and New Zealand Australia      10
```

Other relevant functions: slice, subset, select

Arrange Slice Select

Combining functions

Select: allows you to specific columns.

```
whr15 %>%
  arrange(region, country) %>%
  filter(!is.na(region)) %>%
  select(country, region, starts_with("happin")) %>%
  slice_head()
```

Sort by region and country
Filter those non-missing obs for region if any
Select country, year, and vars that stars with happen
Get the first row

```
## # A tibble: 1 x 4
##   country    region      happiness_rank happiness_score
##   <chr>      <chr>          <dbl>            <dbl>
## 1 Australia Australia and New Zealand     10             7.28
```

Using `ifelse` when creating a variable

We can also create a dummy variable with the `ifelse()` function. The way we use this function is as: `ifelse(test, yes, no)`. We can also use another function called `case_when()`.

```
whr15 %>%
  mutate(
    latin_america_car = ifelse(region == "Latin America and Caribbean", 1, 0)
  ) %>%
  arrange(-latin_america_car) %>%
  head(5)
```

```
## # A tibble: 5 x 9
##   country region  year happiness_rank happiness_score economy_gdp_per~ health_life_exp~ freedom
##   <chr>    <chr>  <dbl>          <dbl>            <dbl>           <dbl>            <dbl>
## 1 Costa ~ Latin~  2015          12        7.23        0.956        0.860        0.634
## 2 Mexico  Latin~  2015          14        7.19        1.02         0.814        0.482
## 3 Brazil  Latin~  2015          16        6.98        0.981        0.697        0.490
## 4 Venezu~ Latin~  2015          23        6.81        1.04         0.721        0.429
## 5 Panama  Latin~  2015          25        6.79        1.06         0.797        0.542
## # ... with 1 more variable: latin_america_car <dbl>
```

Using case_when() to update a variable

Recall the problem we have with regions in the `whr17` data. We can fix it as follows:

```
whr17 <- whr17 %>%
  mutate(
    country = case_when(
      country == "Hong Kong S.A.R., China" ~ "Hong Kong",
      country == "Taiwan Province of China" ~ "Taiwan",
      TRUE ~ country
    )
  )

whr17 %>%
  left_join(regions, by = "country") %>%
  rename(region = region.y) %>%
  select(-region.x) %>%
  select(country, region, everything()) %>%
  filter(is.na(region))
```

Factor variables

Factor variables

- When we imported this data set, we told R explicitly to not read strings as factor.
- We did that because we knew that we'd have to fix the country names.
- The region variable, however, should be a factor.

```
str(whi_panel$region)  
  
##  chr [1:470] "Western Europe" "Western Europe" "Western Europe" "Western Europe" ...
```

Factor variables

To create a factor variable, we use the `factor()` function (or `as_factor()` from the `forcats` package).

- `factor(x, levels, labels)` : turns numeric or string vector `x` into a factor vector.
- `levels`: a vector containing the possible values of `x`.
- `labels`: a vector of strings containing the labels you want to apply to your factor variable
- `ordered`: logical flag to determine if the levels should be regarded as ordered (in the order given).

If your categorical variable does not need to be ordered, and your string variable already has the label you want, making the conversion is quite easy.

Factor variables

Extra exercise: Turn a string variable into a factor.

- Use the mutate function to create a variable called region_cat containing a categorical version of the region variable.
- TIP: to do this, you only need the first argument of the factor function.

How to do it?

```
whr_panel <- mutate(whr_panel, region_cat = factor(region))
```

And now we can check the class of our variable.

```
class(whr_panel$region_cat)
```

```
## [1] "factor"
```

Reshaping a dataset

Reshaping a dataset

Finally, let's try to reshape our dataset using the tidyverse functions. No more `reshape` from Stata. We can use `pivot_wider` or `pivot_longer`. Let's assign our wide format panel to an object called whr_panel_wide.

Long to Wide

Wide to Long

```
whr_panel %>%
  select(country, region, year, happiness_score) %>%
  pivot_wider(
    names_from = year,
    values_from = happiness_score
  ) %>%
  head(3)
```

```
## # A tibble: 3 x 5
##   country     region `2015` `2016` `2017`
##   <chr>       <chr>   <dbl>   <dbl>   <dbl>
## 1 Switzerland Western Europe  7.59    7.51    7.49
## 2 Iceland      Western Europe  7.56    7.50    7.50
## 3 Denmark      Western Europe  7.53    7.53    7.52
```

Reshaping a dataset

Finally, let's try to reshape our dataset using the tidyverse functions. No more `reshape` from Stata. We can use `pivot_wider` or `pivot_longer`. Let's assign our wide format panel to an object called whr_panel_wide.

Long to Wide Wide to Long

```
whr_panel_wide %>%
  pivot_longer(
    cols = `2015`:`2017`,
    names_to = "year",
    values_to = "happiness_score"
  ) %>%
  head(3)
```

```
## # A tibble: 3 x 4
##   country     region     year  happiness_score
##   <chr>       <chr>      <chr>        <dbl>
## 1 Switzerland Western Europe 2015        7.59
## 2 Switzerland Western Europe 2016        7.51
## 3 Switzerland Western Europe 2017        7.49
```

Thank you~