Session 3: Data Processing

R for Stata Users

Rony Rodrigo Maximiliano Rodriguez-Ramirez
The World Bank – DIME | WB Github

24 November 2020

Table of contents

Introduction

Introduction

Goals of this session

- To organize data in a way that it will be easier to analyze it and communicate it.
- We'll use a set of packages that are bundled into something called the `tidyverse`.

Introduction

Goals of this session

- To organize data in a way that it will be easier to analyze it and communicate it.
- We'll use a set of packages that are bundled into something called the `tidyverse`.

Things to keep in mind

- We'll take you through the same steps we've taken when we were preparing the datasets.
- 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. observations are in rows
> 2. variables are in columns
> 3. contained in a single dataset.

Takeaway: long format > wide format

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.

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. Load the tidyverse package.
3. Set your file paths.

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 something don't need to assign everything to an object.

Let's load the tidyverse meta-package:

```
# If you haven't installed the package uncomment the next line
# install.package("tidyverse", dependencies = TRUE)
# install.package("janitor")

library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.0.3

## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0

## Warning: package 'ggplot2' was built under R version 4.0.3

## Warning: package 'tibble' was built under R version 4.0.3

## Warning: package 'tidyr' was built under R version 4.0.3

## Warning: package 'readr' was built under R version 4.0.3

## Warning: package 'purrr' was built under R version 4.0.3
```

File paths

For this session, my file paths are as follows. We will use them to load and export datasets.

```
# Example of my filepaths for this presentation
project           ← "D:/Documents/RA Jobs/DIME/trainings/dime-r-training"

dataWorkFolder    ← file.path(projectFolder,"DataWork")
Data              ← file.path(dataWorkFolder,"DataSets")
finalData         ← file.path(Data,"Final")
rawData           ← file.path(Data,"Raw")
rawOutput         ← file.path(dataWorkFolder,"Output","Raw")
```

If you saved the changes you made to the code during the Intro II session, you can simply run that script.

Loading a dataset in R

Before we start wrangling our data, let's read ourdataset. 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. This is a recap from yesterday's session.

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

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

Loading a dataset in R

Before we start wrangling our data, let's read ourdataset. 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. This is a recap from yesterday's session.

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

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

How to do it?

```
whr15 ← read_csv(file.path(rawData, "Un WHR" , "WHR2015.csv")) %>% clean_names()
whr16 ← read_csv(file.path(rawData, "Un WHR" , "WHR2016.csv")) %>% clean_names()
whr17 ← read_csv(file.path(rawData, "Un WHR" , "WHR2017.csv")) %>% clean_names()
```

> Notice the `clean_names()` function. More on this in the next slide.

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.

> **Pro tip**: Use the `clean_names()` function in a pipe after you load a dataset as we did in the last slide.

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>             <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>
```

Exploring your data

Exploring a data set

Some useful functions from base R:

- `View()`: open the data set
- `class()`: reports object type of 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

Before we explore our data

For this session, we are going to use pipes %>% quite a lot.

- "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 following pipe, and this continues *forever*~~~~

> **Be careful with that forever!!!**

Glimpse your data

This functions give your information about your variables (e.g., type, row, columns,)

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

```
## Rows: 158
## Columns: 12
## $ country                    <chr> "Switzerland", "Iceland", "Denmark", "N ...
## $ region                     <chr> "Western Europe", "Western Europe", "We ...
## $ happiness_rank             <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  ...
## $ happiness_score            <dbl> 7.587, 7.561, 7.527, 7.522, 7.427, 7.40 ...
## $ standard_error             <dbl> 0.03411, 0.04884, 0.03328, 0.03880, 0.0 ...
## $ economy_gdp_per_capita     <dbl> 1.39651, 1.30232, 1.32548, 1.45900, 1.3 ...
## $ family                     <dbl> 1.34951, 1.40223, 1.36058, 1.33095, 1.3 ...
## $ health_life_expectancy     <dbl> 0.94143, 0.94784, 0.87464, 0.88521, 0.9 ...
## $ freedom                    <dbl> 0.66557, 0.62877, 0.64938, 0.66973, 0.6 ...
## $ trust_government_corruption <dbl> 0.41978, 0.14145, 0.48357, 0.36503, 0.3 ...
## $ generosity                 <dbl> 0.29678, 0.43630, 0.34139, 0.34699, 0.4 ...
## $ dystopia_residual          <dbl> 2.51738, 2.70201, 2.49204, 2.46531, 2.4 ...
```

ID variables

ID variables

Desired properties of an ID variable: uniquely and fully identifying.

- An ID variable cannot have duplicates
- An ID variable may never be missing
- The ID variable must be constant across a project
- The ID variable must be anonymous

ID variables

Let's see first:

- **Dimensions of your data**:

```
dim(whr15)
```

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

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

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

ID variables

Let's see first:

- **Dimensions of your data**:

```
dim(whr15)
```

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

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

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

Quick Note:

- Missings 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.

ID variables

In the last example, we used `n_distinct`. This 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.

ID variables

In the last example, we used `n_distinct`. This 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.

Exercise 2: Identify the ID.

Using the `n_distinct` function, can you tell if the following variables are IDs of the whr15 data set? Is any of these variables an ID variable?

1. Region
2. Country

ID variables

In the last example, we used `n_distinct`. This 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.

Exercise 2: Identify the ID.

Using the `n_distinct` function, can you tell if the following variables are IDs of the whr15 data set? Is any of these variables an ID variable?

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
```

ID variables

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
```

ID variables

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
```

```
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

Filter or subsetting a dataset.

```
whr15 %>%
  filter(region == "Western Europe",
         happiness_rank <= 10)
```

```
## # A tibble: 7 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
## # ... with 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()`
- Filter only for the regions: (1) Eastern Asia and (2) North America.

dplyr::filter

Exercise 3: Filter the dataset.

- Use `filter()`
- Filter only for the regions: (1) Eastern Asia and (2) North America.

This would be the normal way to do it:

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

dplyr::filter

Exercise 3: Filter the dataset.

- Use `filter()`
- Filter only for the regions: (1) Eastern Asia and (2) North America.

This would be the normal way to do it:

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

A more elegant approach would be:

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

dplyr::filter regular expressions

One advantage of the filter command over Stata is that you can also integrate regular expressions in quite a better way. Let's say that we want to subset all regions' divisions that have `America` in their names. We can use the following:

```
whr15 %>%
  filter(grepl("America", region)) %>%
  head(2)
```

```
## # A tibble: 2 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per~
##   <chr>   <chr>           <dbl>           <dbl>          <dbl>            <dbl>
## 1 Canada  North~              5            7.43         0.0355            1.33
## 2 Costa ~ Latin~             12            7.23         0.0445            0.956
## # ... with 6 more variables: family <dbl>, health_life_expectancy <dbl>,
## #   freedom <dbl>, trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

dplyr::filter regular expressions

One advantage of the filter command over Stata is that you can also integrate regular expressions in quite a better way. Let's say that we want to subset all regions' divisions that have `America` in their names. We can use the following:

```
whr15 %>%
  filter(grepl("America", region)) %>%
  head(2)
```

```
## # A tibble: 2 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per~
##   <chr>   <chr>           <dbl>           <dbl>          <dbl>            <dbl>
## 1 Canada  North~              5            7.43         0.0355             1.33
## 2 Costa ~ Latin~             12            7.23         0.0445             0.956
## # ... with 6 more variables: family <dbl>, health_life_expectancy <dbl>,
## #   freedom <dbl>, trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

> Notice that I have used `head()` to show just the first 2 observations of the subset. If you want to save this subset you can assign it to an objet. For example `whr15_east <- + the code above`. The arrow (<-) remembert that is a mix of < and -. The font we are using is called Fira Code which finds the two characters (a ligature).

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 return the value of true, otherwise, return to a value of 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 missings values in a specif variable.

The function syntax in a pipeline is as follows:

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

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 return the value of true, otherwise, return to a value of 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 missings values in a specif variable.

The function syntax in a pipeline is as follows:

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

What are we returning here?

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 return the value of true, otherwise, return to a value of 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 missings values in a specif 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>             <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
## # ... 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., !
> In case we want to keep the observations that contains missing information we will only use `is.na()`.

Other relevant functions: slice, subset, select

`Arrange`: allows you to order by a specific column.

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

```
## # A tibble: 5 x 12
##   country region happiness_rank happiness_score standard_error economy_gdp_per~
##   <chr>   <chr>           <dbl>           <dbl>          <dbl>             <dbl>
## 1 Austra~ Austr~             10            7.28         0.0408             1.33
## 2 New Ze~ Austr~              9            7.29         0.0337             1.25
## 3 Albania Centr~             95            4.96         0.0501             0.879
## 4 Armenia Centr~            127            4.35         0.0476             0.768
## 5 Azerba~ Centr~             80            5.21         0.0336             1.02
## # ... with 6 more variables: family <dbl>, health_life_expectancy <dbl>,
## #   freedom <dbl>, trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

Creating new variables

Creating new variables

In the tidyverse, we refer to creating variables as mutating

So, instead of **gen**erate, we use `mutate()`. Let's say we want to have interactions:

```
whr15 %>%
  arrange(region, country, -happiness_rank) %>%
  mutate(
    hap_hle = happiness_score * health_life_expectancy,
  ) %>%
  select(country:happiness_score, health_life_expectancy, hap_hle) %>%
  head(5)
```

```
## # A tibble: 5 x 6
##   country   region      happiness_rank happiness_score health_life_expe~ hap_hle
##   <chr>     <chr>                <dbl>           <dbl>              <dbl>   <dbl>
## 1 Australia Australia ~             10            7.28              0.932    6.79
## 2 New Zeal~ Australia ~              9            7.29              0.908    6.62
## 3 Albania   Central an~             95            4.96              0.813    4.03
## 4 Armenia   Central an~            127            4.35              0.730    3.18
## 5 Azerbaij~ Central an~             80            5.21              0.640    3.34
```

Creating new variables: Dummy variables

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

Q Well, what do you think is happening to this variable?

Creating new variables: Dummy variables

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

Q Well, what do you think is happening to this variable?

A The variable we created contains either TRUE or FALSE.
If we want to have it as a numeric (1 or 0), we could include `as.numeric()`

Creating new variables: Dummy variables

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

Q Well, what do you think is happening to this variable?

A The variable we created contains either TRUE or FALSE.
If we want to have it as a numeric (1 or 0), we could include `as.numeric()`

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

Creating new variables: Dummy variables

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

Q Well, what do you think is happening to this variable?

A The variable we created contains either TRUE or FALSE.
If we want to have it as a numeric (1 or 0), we could include `as.numeric()`

```
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)))
  )
```

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 13
##   country region happiness_rank happiness_score standard_error economy_gdp_per~
##   <chr>   <chr>           <dbl>           <dbl>          <dbl>             <dbl>
## 1 Costa ~ Latin~             12            7.23         0.0445             0.956
## 2 Mexico  Latin~             14            7.19         0.0418             1.02
## 3 Brazil  Latin~             16            6.98         0.0408             0.981
## 4 Venezu~ Latin~             23            6.81         0.0648             1.04
## 5 Panama  Latin~             25            6.79         0.0491             1.06
## # ... with 7 more variables: family <dbl>, health_life_expectancy <dbl>,
## #   freedom <dbl>, trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>, latin_america_car <dbl>
```

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

Let's imagine now that we want to create a variable at the region level -- recal `bys gen` in Stata.In R, we can use `group_by()` before we mutate. So for this example, we going to combine the chain 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`.

Creating variables by groups

Let's imagine now that we want to create a variable at the region level -- recal `bys gen` in Stata.In R, we can use `group_by()` before we mutate. So for this example, we going to combine the chain 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`.

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

Creating multiple variables at the same time

With the new version of `dplyr`, we now can create multiple variables in an easier 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?

- In R we can use the new 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")

whr15 %>%
  group_by(region) %>%
  summarize(
    across(all_of(vars), mean)
  )
```

Creating variables

Exercise 5: Create a variable called year that equals to the year of each dataframe .

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

Creating variables

Exercise 5: 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( ... )

bind_rows( ... )
```

Exercise 6: Append data sets.

- Use either functions to append the three WHR datasets:

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( ... )

bind_rows( ... )
```

Exercise 6: Append data sets.

- Use either functions to append the three WHR datasets:

How to do it?

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

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( ... )

bind_rows( ... )
```

Exercise 6: Append data sets.

- Use either functions 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?

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( ... )

bind_rows( ... )
```

Exercise 6: Append data sets.

- Use either functions 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, columns compatibility is quite junky.

Appending and merging data sets

To be honest, the most important variable that we will need to include in the 2017 dataset is the region variable.

Exercise 7: Fixing our variables and appending the dfs correctly.

Exercise 7a:

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

Appending and merging data sets

To be honest, the most important variable that we will need to include in the 2017 dataset is the region variable.

Exercise 7: Fixing our variables and appending the dfs correctly.

Exercise 7a:

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

```
regions ← readRDS(file.path(Data, "Raw", "Un WHR", "regions.RDS"))
```

Appending and merging data sets

We can use the `left_join()` function 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.

Appending and merging data sets

We can use the `left_join()` function 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.

Exercise 7b:

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

Appending and merging data sets

We can use the `left_join()` function 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.

Exercise 7b:

- 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 use to **order** columns!

Appending and merging data sets

Exercise 7c:

Check if there is any other country without region info:

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

Appending and merging data sets

Exercise 7c:

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
##   <chr>   <chr>           <dbl>           <dbl>        <dbl>       <dbl>
## 1 Taiwan~ <NA>               33            6.42         6.49        6.35
## 2 Hong K~ <NA>               71            5.47         5.55        5.39
## # ... with 8 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>, generosity <dbl>,
## #   trust_government_corruption <dbl>, dystopia_residual <dbl>, year <dbl>
```

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?

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.

Appending and merging data sets

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

Exercise 8: 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., 15,16,16.

Appending and merging data sets

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

Exercise 8: 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., 15,16,16.

```
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
```

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 save 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, row.names = TRUE)`:

- `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")
- `row.names:` by default, R adds a column to the CSV file with the names (or numbers) of the rows in the data frame. Set it to `FALSE` if you don't want that column to be exported.

Saving a dataset

Exercise 9: Save the dataset.

- Use `write.csv()`
- Use `file.path()`

```
# Save the whr data set

write.csv(whr_panel,
          file.path(finalData,"whr_panel.csv"),
          row.names = FALSE)
```

Saving a dataset

Exercise 9: Save the dataset.

- Use `write.csv()`
- Use `file.path()`

```
# Save the whr data set

write.csv(whr_panel,
          file.path(finalData,"whr_panel.csv"),
          row.names = FALSE)
```

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

Saving a dataset

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

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

```
# Save the data set

saveRDS(whr_panel, file = file.path(finalData, "whr_panel.Rds"))
```

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(whr_panel$region)
```

```
##  chr [1:470] "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).

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

Exercise 10: 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.

Factor variables

Exercise 10: 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))
```

Factor variables

Exercise 10: 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(5)
```

```
## # A tibble: 5 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
## 4 Norway      Western Europe    7.52   7.50   7.54
## 5 Canada      North America     7.43   7.40   7.32
```

Thank you~~