Session 6: R Programming Practices

R for Stata Users

Luiza Andrade, Rob Marty, Rony Rodriguez-Ramirez, Luis Eduardo San Martin, Leonardo Viotti
The World Bank | WB Github

April 2021

Table of contents

Introduction

What is this session about?

- In the previous sessions, you learned how to work with R
- You are probably eager to start programming in R by now
- But before you start, we recommend learning how to write R code that will be **reproducible, efficient, intelligible and easy to navigate**
- Indeed, that's what this session is about!

Introduction

What this session is about

- We will cover common coding practices in R so that you can make **the most efficient use** for it
- We will also discuss some styling conventions to make your code **readable and reproducible**
- This will give you a solid foundation to write code in R and hopefully you'll be able to skip some painful steps of the "getting-your-hands-dirty" learning approach

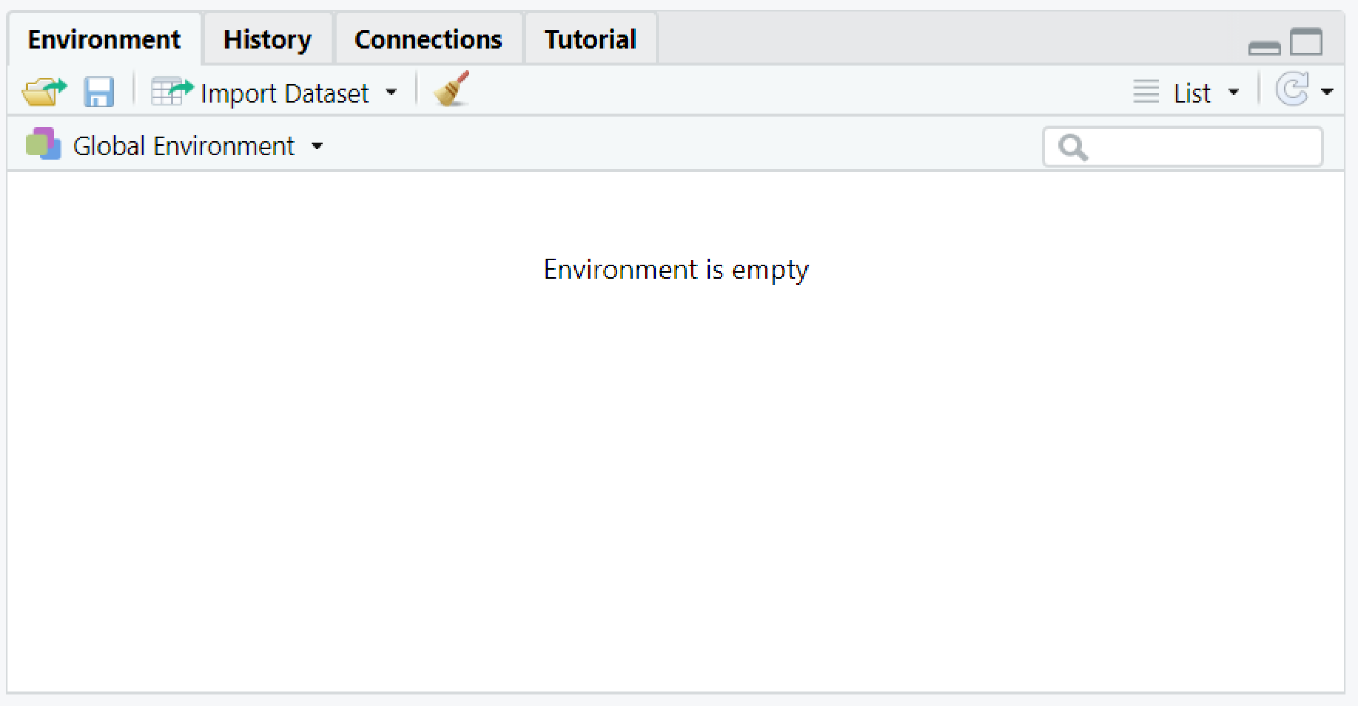
Initial settings

Initial settings

- Let's start by opening RStudio or by closing and opening it again

- Notice two things:

1. Your environment is *probably* empty (it's OK if it's not)



Initial settings

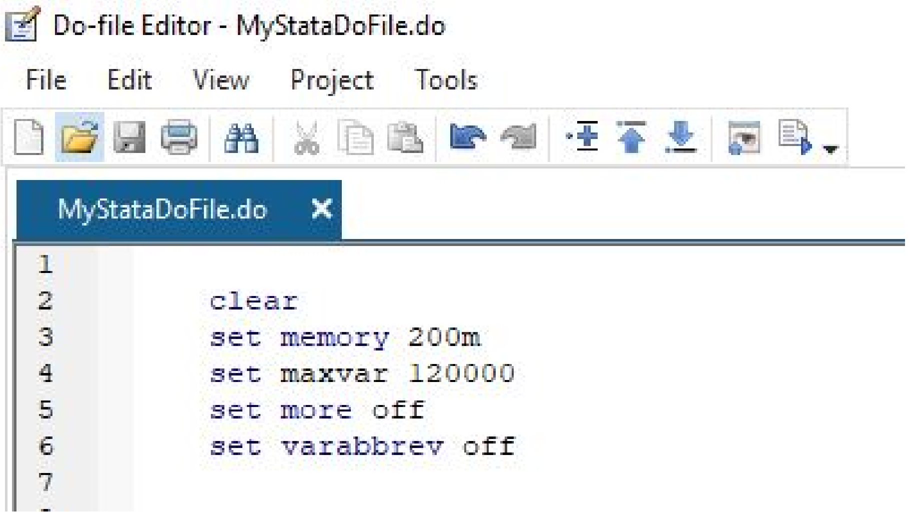
2- Go to the `Console` panel and use the up and down keys to navigate through previously executed commands. They are saved by default in a file named `.Rhistory` that you might have noticed

Name	Date modified	Type
.git	4/6/2021 2:07 PM	File folder
.Rproj.user	4/6/2021 9:51 AM	File folder
DataWork	4/5/2021 4:37 PM	File folder
Presentations	4/6/2021 5:16 PM	File folder
.gitignore	4/5/2021 4:37 PM	GITIGNORE File
.Rhistory	4/6/2021 4:18 PM	RHISTORY File
dime-r-training.Rproj	4/6/2021 4:17 PM	R Project
LICENSE	12/15/2020 2:53 PM	File
README.md	4/5/2021 4:37 PM	MD File

- We'd usually want these two things -- an **empty environment** and the **history of commands** executed in previous sessions -- to be present every time we open a new RStudio session

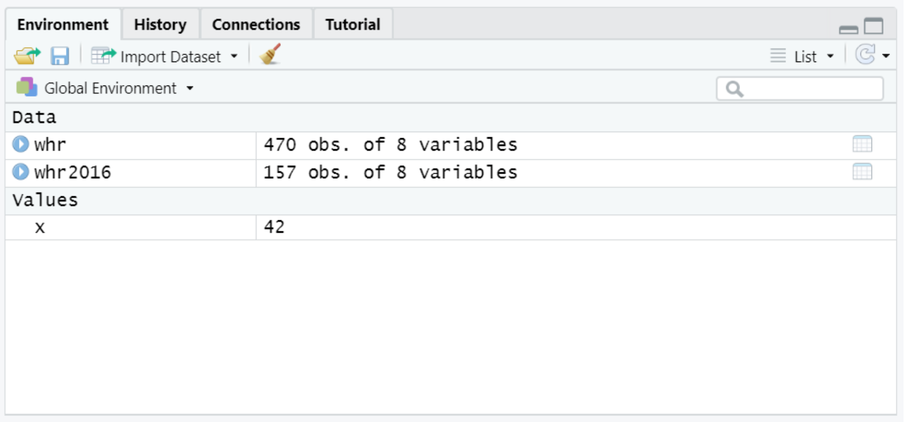
Initial settings

Have you ever seen these lines of code before?



R Initial settings

- We **don't need to set the memory or the maximum number of variables** in R
- The equivalent of `set more off` is the default
- The equivalent of `clear all` is not a default setting, but we'll change that in exercise 1
- In any case, remember that you can see all the objects in your computer's memory at any point in the `Environment` panel

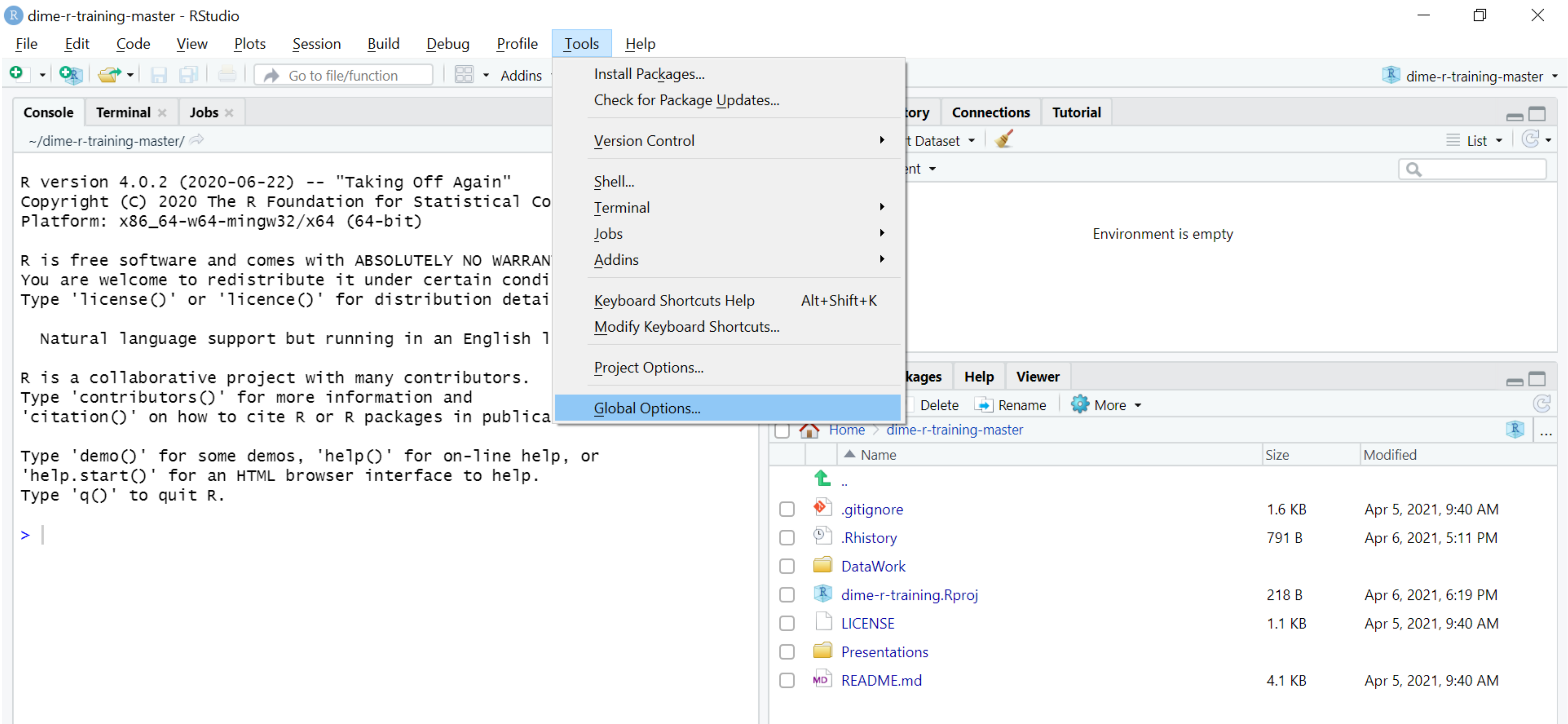


Initial settings

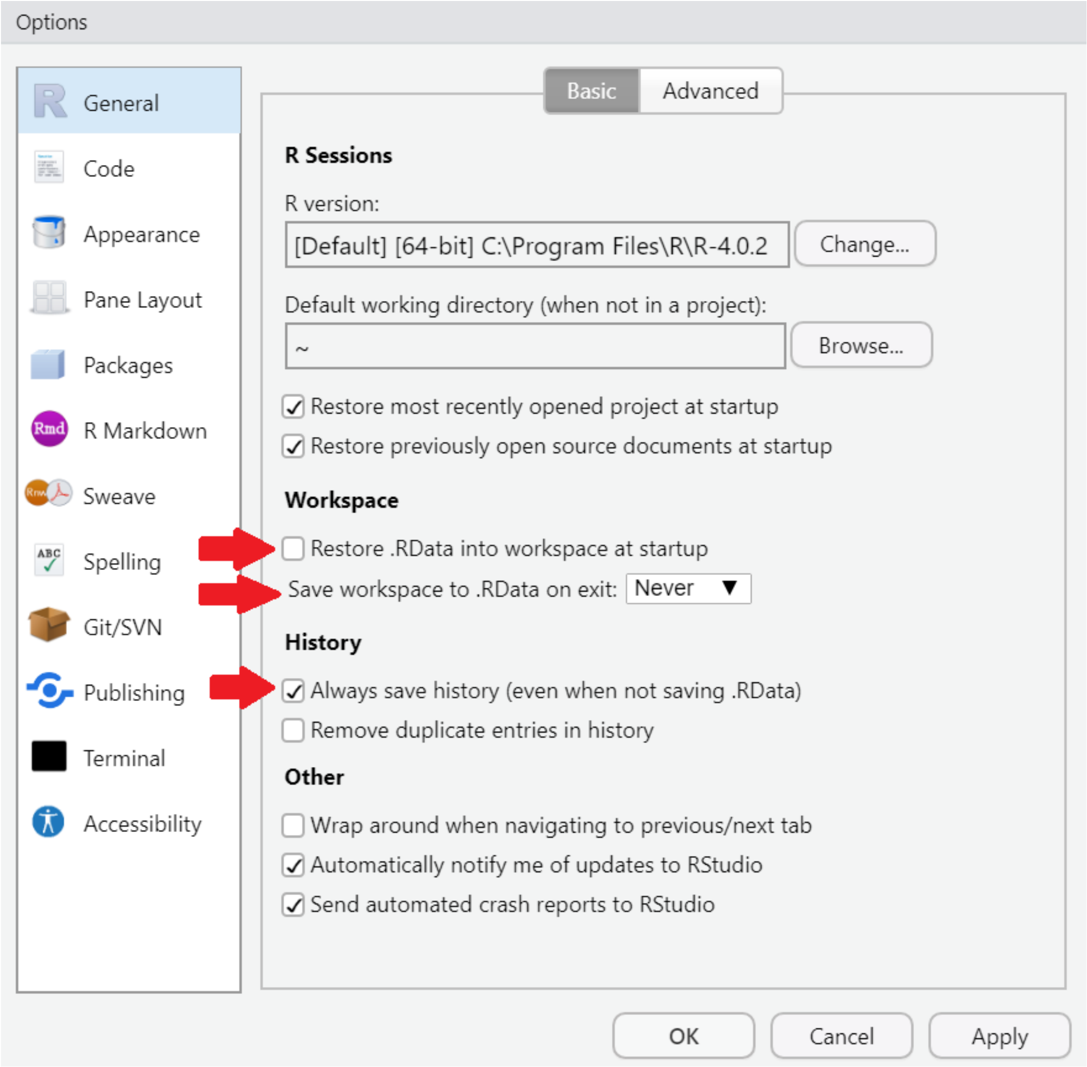
Exercise 1: you'll never have to use the equivalent of `clear all`

1. Go to `Tools` > `Global Options...`
2. In the `General` tab, make sure the following options are set:
   - Un-check *Restore .RData into workspace at startup*
   - For *Save workspace to .RData on exit*, select *Never*
   - Make sure *Always save history (even when not saving .RData)* is checked
3. Now restart RStudio

Initial settings



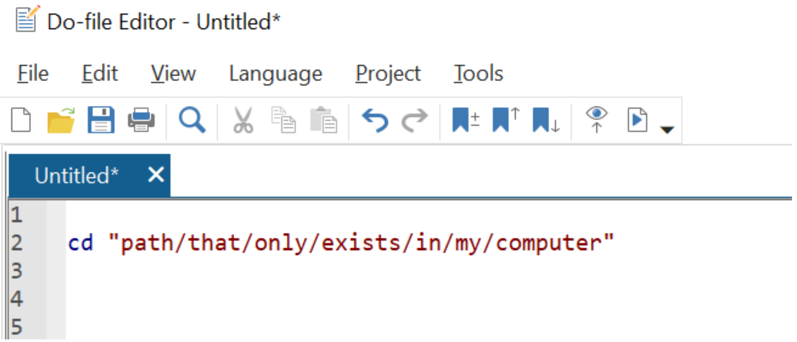
Initial settings



File paths

File paths

- What about working directories? We usually do something like this every time we start a new script in Stata:



- The direct equivalent to `cd` in R is this command:

```
setwd("your/path")
```

- However, we recommend not using it unless it's absolutely necessary (never, if possible)

File paths

- Instead, you should use RStudio projects and the `here` library
- **Important:** We won't get into the specifics of directory organization here, but we'll assume that all the files you use for a specific project (data, scripts, and outputs) reside in the same project directory. We'll call this the **working directory**

RStudio Projects

- RStudio projects let you "bind" your project files to a root directory, regardless of the path to it
- This is crucial because it allows smooth interoperability between different computers where the exact path to the project root directory differs
- Additionally, each RStudio project you work on keeps their own history of commands!

RStudio projects

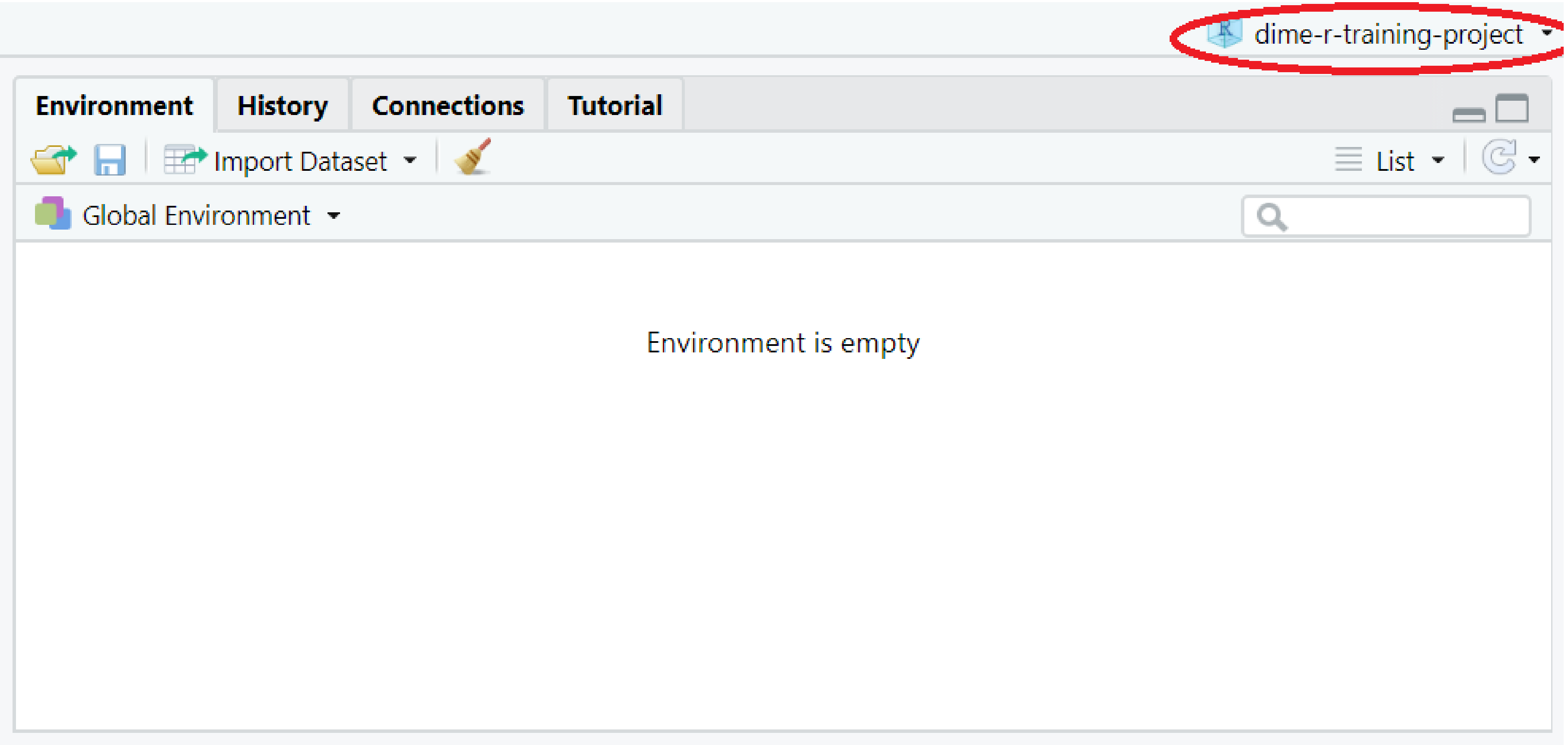
Exercise 2: Create a new RStudio project

Follow these instructions:

1. On RStudio, select `File` > `New Project...`
2. Select `New Directory` > `New Project`
3. Assign the name: `dime-r-training-project` to the project

File paths

RStudio projects



File paths

The `here` library

- `here` locates files relative to your project root
- It uses the root project directory to build paths to files easily
- Similar to RStudio projects, it allows for interoperability between different computers where the absolute path to the same file is not the same

File paths

Usage of `here`

1. Load the `here` library:

```
library(here)
```

1. Now you'll be able to use `here()` to point the location of every file relative to your project root

   - For example, to load a `csv` file located in: `C:/WBG/project-root-name/data/raw/data-file.csv`, you should use:

```
df <- read.csv(here("data", "raw", "data-file.csv"))
```

- **Note:** Your project root is the directory that contains the `.Rproj` file

File paths

Exercise 3: Combining `here` and RStudio projects

1. Go to the OSF page of the course: https://osf.io/86g3b/
2. Download the file in: `R for Stata Users - April 2021` > `Data` > `DataWork.zip`
3. Unzip the file in your RStudio project root folder
   - This is the folder where the file `dime-r-training-project.Rproj` sits
   - If you didn't change the default directory when creating the RStudio project, it will be in your `Documents` folder in Windows or in your Home directory in Mac or Linux
4. On RStudio, go to `File` > `New File` > `R Script`

File paths

Exercise 3: Combining `here` and RStudio projects

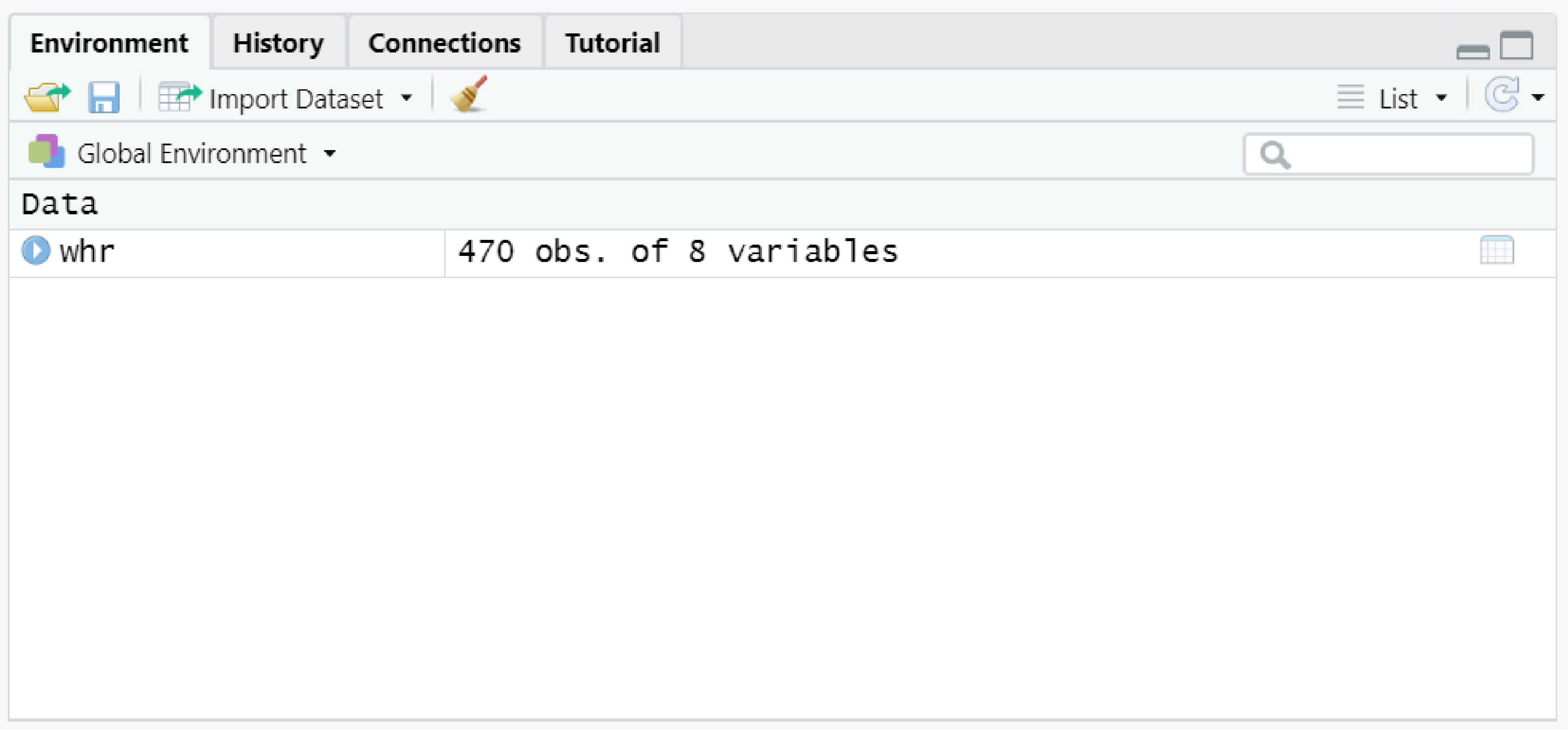
5- Save this new empty script in `DataWork` > `Code` as `exercises.R`

6- Now let's test if that worked. Load the `here` library read the `csv` file `DataWork/DataSets/Final/whr_panel.csv` using the function `here()`:

```
library(here)
whr <- read.csv(here("DataWork", "DataSets", "Final", "whr_panel.csv"))
```

RStudio projects and `here`

If you did the exercise correctly, you should see the `whr` data frame listed in the Environment panel



Creating a document outline in RStudio

Creating a document outline in RStudio

- RStudio also allows you to **create an interactive index** for your scripts
- To add a section to your code, create a commented line with the title of your section and add at least 4 trailing dashes (`----`), pound signs (`####`) or equal signs (`====`) after it

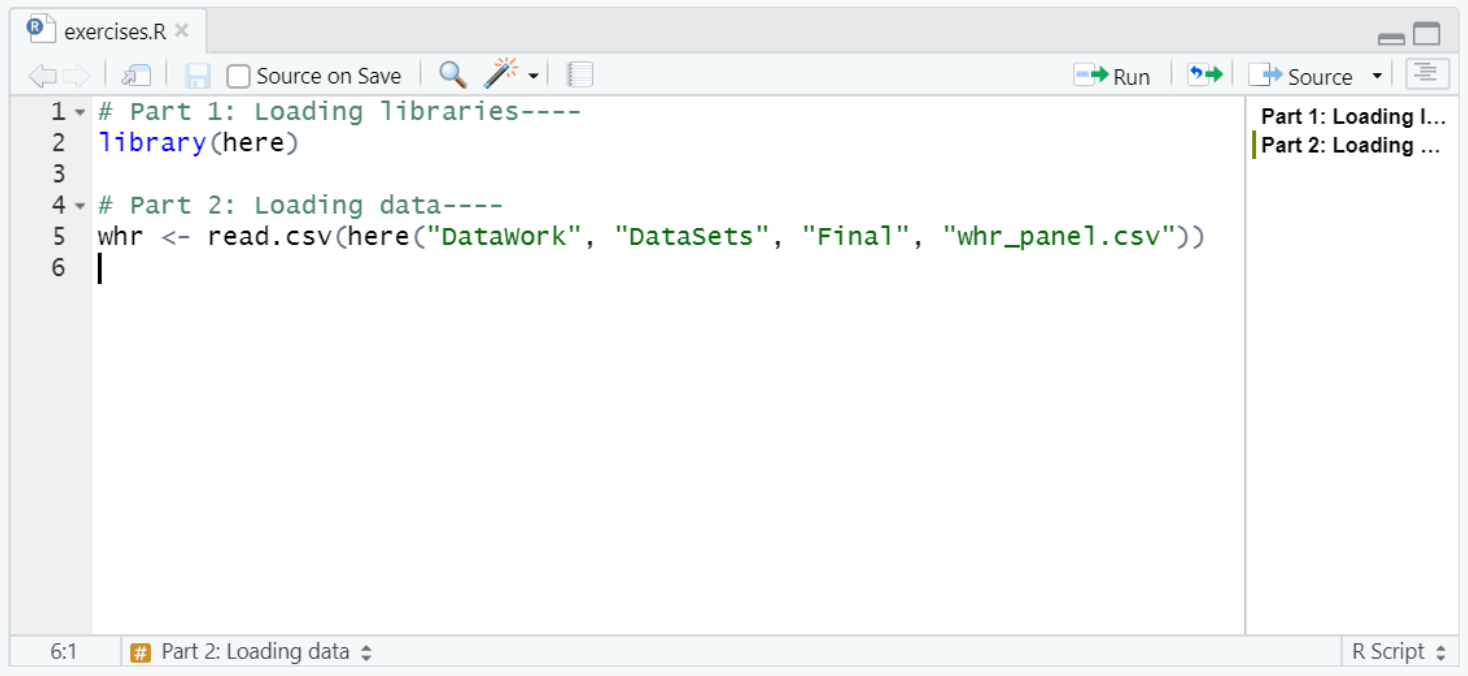
Creating a document outline in RStudio

Exercise 4: Headers

1. In your script, add a header before the line where you used `library(here)` with the text: `# Part 1: Loading libraries----`
2. Add the following header before `read.csv(...)`: `Part 3: Loading data----`
   - Remember: you create a section header by adding at least 4 trailing dashes (`-`), pound (`#`) or equal (`=`) signs in a comment line
3. Note that once you create a section header, an arrow appears right next to the row number. Click on the arrows to see what happens.

Creating a document outline in RStudio

- The outline can be accessed by clicking on the button on the top right corner of the script window. You can use it to jump from one section to another
- You can also use the keyboard shortcuts `Alt + L` (`Cmd + Option + L` on Mac) and `Alt + Shift + L` to collapse and expand sections



Using packages

Using packages

Packages

- Since there is a lot of people developing for R, it can have many different functionalities.
- To make it simpler, these functionalities are bundled into packages.
- A package is just **a unit of shareable code**.

Using packages

Packages

- It may contain new functions, but also more complex functionalities, such as a Graphic User Interface (GUI) or settings for parallel processing (similar to Stata MP).
- They can be shared through R's official repository - CRAN (13,000+ packages reviewed and tested).
- There are many other online sources such as GitHub, but it's important to be careful, as these probably haven't gone through a review process as rigorous as those in CRAN.

Packages

- To install and use packages you can either do it with the user interface or by the command prompt.

```
# Installing a package
install.packages("tidyverse",
                 dependencies = TRUE)
# the dependencies argument also installs all other packages
# that it may depend upon to run
```

- You only have to install a package once, but you have to **load it every new session**.

Packages

Exercise 5

Add:

```
library(tidyverse)
```

to Part 2 in your script and run it

Using packages

Warnings vs errors

What if this happens?

```
> library(tidyverse)
-- Attaching packages --------------------------------------------- tidyverse 1.3.0 --
v ggplot2 3.3.2     v purrr   0.3.4
v tibble  3.1.0     v dplyr   1.0.5
v tidyr   1.1.2     v stringr 1.4.0
v readr   1.3.1     v forcats 0.5.0
-- Conflicts ------------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()

> # Part 3: Loading data----
> whr <- read.csv(here("DataWork", "DataSets", "Final", "whr_panel.csv"))
Warning messages:
1: package ‘tidyverse’ was built under R version 4.0.3
2: package ‘tibble’ was built under R version 4.0.4
3: package ‘dplyr’ was built under R version 4.0.4
```

Warnings vs errors

R has two types of error messages, `warnings` and actual `errors`:

- `Errors` - break your code, usually preventing it from running.
- `Warnings` - usually mean that nothing went wrong yet, but you should be careful.

RStudio's default is to print warning messages, but not to stop the code at the lines where they occur. You can configure R to stop at warnings if you want.

Functions inception

Functions inception

A function inside a function

- In R, you can **write one function inside another**
- In fact, you have already done this several times in this course
- Here's an example:

Functions inception

```
# Print the summary of the logarithm of the happiness score

## The long way:
log_score <- log(whr$happiness_score)
summary(log_score)

# The shortcut
summary(log(whr$happiness_score))
```

Functions inception

A function inside a function

- This is a simple example of **metaprogramming** (that's the real name of this technique) and may seem trivial, but it's not
- For starters, you can't do it in Stata!

Functions inception

```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   15.1   Copyright 1985-2017 StataCorp LLC
  Statistics/Data Analysis            StataCorp
                                      4905 Lakeway Drive
     MP - Parallel Edition            College Station, Texas 77845 USA
                                      800-STATA-PC        http://www.stata.com
                                      979-696-4600        stata@stata.com
                                      979-696-4601 (fax)

681-user 4-core Stata network perpetual license:
       Serial number:  501506002486
         Licensed to:  WBG User
                       World Bank Group

Notes:
      1.  Unicode is supported; see help unicode_advice.
      2.  More than 2 billion observations are allowed; see help obs_advice.
      3.  Maximum number of variables is set to 120000; see help set_maxvar.
      4.  New update available; type -update all-

running C:\Program Files (x86)\Stata15\sysprofile.do ...

. sysuse auto
(1978 Automobile Data)

. summarize log(make)
variable log not found
r(111);

.
```

Functions inception

Piping

- This is a **very powerful technique**, as you will soon see
- It's **also a common source of error**, as you can only use one function inside the other if the output of the inner function is the same as the input of the outer function
- It can also get quite tricky to follow what a line of code with multiple functions inceptions is doing
- And that is why we use pipes: `%>%`

Functions inception

Piping

```
# 1: Doing it the long way ----------------------------
log_score <- log(whr$happy_score)
mean(log_score)

# 2: Shortcut to get to the same place -----------------
mean(log(whr$happy_score))

# 3: Now with pipes ------------------------------------
whr$happy_score %>%
  log() %>%
  mean()
```

In a few words:

- `x %>% f()` is the same as `f(x)`
- `x %>% f() %>% g()` is the same as `g(f(x))`

Functions inception

Piping

Now that you know piping exists in R, you should know that it can **drastically improve code readability**. And from now on you can also laugh if you see this in some tidyverse nerd laptop sticker or t-shirt:

Looping

Loops

- In Stata, we use `for` loops quite a lot
- The equivalent to that in R would be to write a `for` loop like this

```r
# A for loop in R
for (number in 1:5) {
    print(number)
}
```

Looping

```r
for (number in 1:5) {
    print(number)
}
```

Code Start Over Run Code

Column extraction operators

- Remember the use of `$` to extract columns from a dataframe?
- Other than `$`, we can also use double brackets to extract the column of a dataframe:

```
# With $:
whr$year
```

```
# With [[]]:
whr[["year"]] # Notice the use of double quotes
```

Column extraction operators: `[[]]` vs `$`

What's the key difference between them?

Well, `[[]]` lets us use other objects to refer to column names, while `$` doesn't

```
col_name <- "year"
head(whr$col_name) # this returns a NULL object because no column has the name "col_name" in whr
```

```
## Warning: Unknown or uninitialised column: `col_name`.
```

```
## NULL
```

```
col_name <- "year"
head(whr[[col_name]])
```

```
## [1] 2015 2015 2015 2015 2015 2015
```

Looping

Column extraction operators: `[[]]` vs `$`

This difference is key because we can use `[[]]` to loop through column names, while this is not directly possible with `$`.

```r
# Printing the first observation of every column of whr
for (col in colnames(whr)) {
  whr[[col]] %>%
    head(1) %>%
    print()
}
```

```
## [1] "Switzerland"
## [1] "Western Europe"
## [1] 2015
## [1] 1
## [1] 7.587
## [1] 1.39651
## [1] 0.94143
## [1] 0.66557
```

Looping

Apply

- Using `[[]]` to loop through columns works very similar to how we usually use `for` loops in Stata
- R, however, has a set of functions that allows users to loop through an object **in a more efficient way**, without using explicit loops
- They're called `apply` and there are many of them, with different use cases
- If you look for the `apply` help file, you can see all of them
- For the purpose of this training, we will only use two of them, `sapply` and `apply`

Apply

- The syntax of `sapply()` is:

```
sapply(X, FUN, ...)
```

- Its main arguments are:
  - **X:** a data frame, matrix or vector the function will be applied to
  - **FUN:** the function you want to apply
- `sapply()` applies the function (`FUN`) to all the elements of `X`. If `X` is a data frame then the function is applied column-wise, while if it's a vector or a list it is applied item-wise
- The output of `sapply()` is a vector with the results

Looping

```r
# A for loop in R
for (number in c(1.2, 2.5)) {
  print(round(number))
}

# A much more elegant option
sapply(c(1.2, 2.5), round)
```

Looping

```r
# Printing the first observation of every column of whr
for (col in colnames(whr)) {
  print(head(whr[[col]], 1))
} # Option 1

sapply(whr, head, 1) # A more elegant and efficient option
```

Loops vs Apply

- When looping, you repeat the same operation over a set of items
- `apply()`, instead, takes all your elements at once and applies an operation to them simultaneously
- The difference is like this:
    - Imagine you ask a yes/no question to a group of people
    - You can collect the answers by asking each one of them individually -- **this is looping**
    - Otherwise, you can ask them to raise their hands and collect all answers at once -- **this is** `apply()`
- The output of a loop is the regular output of the operation you're repeating, times the number of iterations you did
- The output of `apply()` will be a vector most of times, but it can also be a list or an array of elements (a matrix)

Apply

- A more general version of `sapply()` is the `apply()` function. This is its syntax:

```
apply(X, MARGIN, FUN, ...)
```

- Arguments:
  - **X:** a data frame (or matrix) the function will be applied to
  - **MARGIN:** 1 to apply the function to all rows or 2 to apply the function to all columns
  - **FUN:** the function you want to apply
- `apply()` applies a function (`FUN`) to all columns or rows of matrix (`X`). A value of 1 in `MARGIN` indicates that the funcion should be applied row-wise, while 2 indicates columns

Looping

```r
matrix <- matrix(c(1, 24, 9, 6, 9, 4, 2, 74, 2), nrow = 3) # Defining a matrix
apply(matrix, 1, mean) # row means
```

```
## [1]  3.00000 35.66667  5.00000
```

```r
apply(matrix, 2, mean) # column means
```

```
## [1] 11.333333  6.333333 26.000000
```

Custom functions

Custom functions

Writing your own functions

- As we have said several times, **R is super flexible**
- One example of that is that it's **super easy and quick to create custom functions**
- Here's how:

Custom functions

```
square <- function(x) {
  y <- x ^ 2
  return(y)
}
```

Code   Start Over   Run Code

```

```

Custom functions

Exercise 6: Create a function

Create a function named `zscore` that standardizes the values of a vector.

- Recall the outline of functions in R:

```
function_name <- function(input) {

  output <- operation(input)

  return(output)

}
```

- Hints:
  - The command to obtain the mean of a vector is `mean(x)`
  - The command to get the SD of a vector is `sd(x)`
  - R is vectorized: you can operate vectors and number directly and the result will be a vector
  - Don't forget to include the argument `na.rm = TRUE` in `mean()` and `sd()`

Custom functions

```r
zscore <- function(x) {
    mean <- mean(x, na.rm = TRUE)
    sd   <- sd(x, na.rm = TRUE)
    z    <- (x - mean)/sd
    return(z)
  }
```

Custom functions

Exercise 7: Putting it all together!

1. Use tidyverse's `select()` to select the columns `health_life_expectancy` and `freedom` in `whr`

2. Use `sapply()` combined with the `zscore` function to get the z-score of these two columns

- Hint:
    - Use the pipes (`%>%`) successively
    - Remember that we don't use parenthesis next to the function name we're using `sapply()` with

Custom functions

```
whr %>%
  select(health_life_expectancy, freedom) %>%
  sapply(zscore)
```

Code | Start Over | Run Code

```
1
2
3
```

Indentation

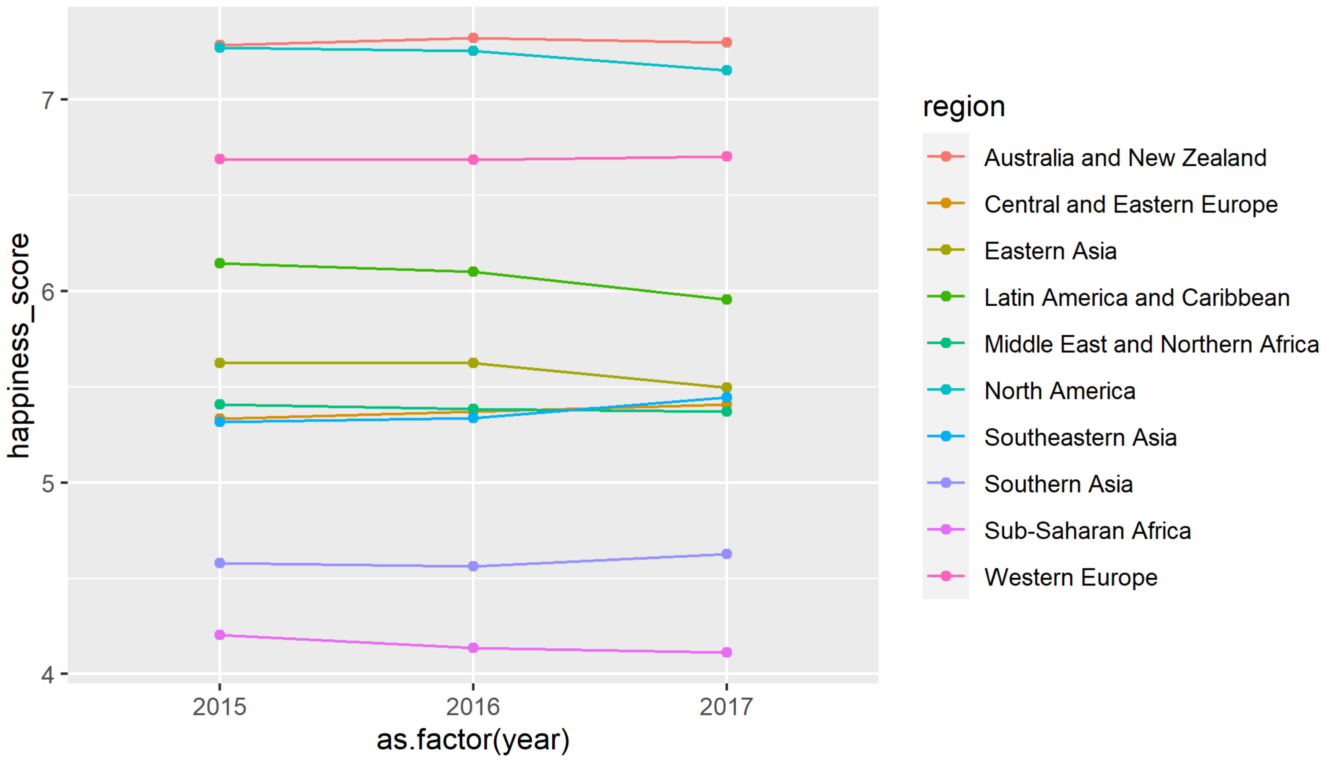
Indentation

```
# Here's some code
annualHappy_reg <- aggregate(happy_score ~ year + region, data = whr, FUN = mean)
ggplot(annualHappy_reg,aes(y = happy_score,x = as.factor(year), color = region, group = region)) +
geom_line() + geom_point()
```

```
# Here's the same code
annualHappy_reg <-
  aggregate(happiness_score ~ year + region,
            data = whr,
            FUN = mean)

ggplot(annualHappy_reg,
       aes(y = happiness_score,
           x = as.factor(year),
           color = region,
           group = region)) +
geom_line() +
geom_point()
```

Indentation



Indentation

Why indent?

- R understands what unindented code says, but it can be **quite difficult for a human being to read it**
- On the other hand, **white space does not have a special meaning for R**, so it will understand code that is more readable for a human being

Indentation

Why indent?

- Indentation in R looks different than in Stata:
  - To indent a whole line, you can select that line and press `Tab`
  - To unindent a whole line, you can select that line and press `Shift + Tab`
  - However, this will not always work for different parts of a code in the same line
- In R, we typically don't introduce white space manually
- It's rather introduced by RStudio for us

Exercise 8: Indentation in R

To see an example of how indenting works in RStudio, let's go back to our first example with `sapply()`:

```
# A much more elegant loop in R
sapply(c(1.2,2.5), round)
```

1. Add a line between the two arguments of the function (the vector of numbers and `round`)

2. Now add a line between the numbers in the vector.

Indentation

Note that RStudio formats the different arguments of the function differently:

```r
# A much more elegant loop in R
sapply(c(1.2,
         2.5),
       round)
```

Thank you!

Appendix

Appendix - Initial settings

`.Rhistory` and `.RData`

- `.Rhistory` automatically stores the commands entered in the console
- `.RData` stores the objects in your environment only if you save your workspace, and loads them again in the next RStudio session
- Both files are relative to the working directory where your RStudio session started

Assignment 1

Create a function that

1. Takes as argument a vector of packages names
2. Loops through the packages listed in the input vector
3. Install the packages
4. Loads the packages

If statements

- Installing packages can be time-consuming, especially as the number of packages you're using grows, and each package only needs to be installed once
- We often use locals in Stata to create section switches to install packages
- In R, the equivalent to that would be to create a new object as a section switch

Exercise: Creating an if statement

- Create a scalar object called PACKAGES and assign it any numerical value.
    - TIP: Section switches can also be Boolean objects.
- Now create an if statement using this switch scalar as a switch to indicate if you want to install a set of packages:

```r
# Turn switch on
PACKAGES <- 1

# Install packages
if (PACKAGES == 1) {
  install.packages(packages,
                   dependencies = TRUE)
}
```

If statements

Possible variations would include

```
# Turn switch on
PACKAGES <- TRUE

# Using a Boolean object
if (PACKAGES == TRUE) {
  install.packages(packages, dep = TRUE)
}

# Which is the same as
if (PACKAGES) {
  install.packages(packages, dep = TRUE)
}
```

Appendix - Assignment 2

Exercise: Create a function that...

1. Takes as argument a vector of packages names
2. Loops through the packages listed in the input vector
3. Tests if a package is already installed
4. Only installs packages that are not yet installed
5. Loads the packages

- TIP: to test if a package is already installed, use the following code:

```
# Test if object x is contained in
# the vector of installed packages
x %in% installed.packages()
```

Appendix - Other file path practices

File paths best practices

- RStudio projects are a nice option when all your project files sit in the same root folder
- But what happens if that's not the case? For example: if your data is in Dropbox or OneDrive and your code is in a GitHub repository folder
- If that happens, we at DIME Analytics recommend always using **explicit** and **dynamic** file paths
  - **Explicit** means you're explicitly stating where the file will be saved -- instead of setting the working directory, for example
  - **Dynamic** means that you don't need to adjust every file path in the script when you change from one machine to another -- they're updated based on a single line of code to be changed

Appendix - File paths best practices

Explicit and dynamic file paths:

```r
# Define dynamic file path
finalData <- "C:/Users/luiza/Documents/GitHub/
              dime-r-training/
              DataWork/DataSets/Final"

# Load data set
whr <- read.csv(file.path(finalData,"whr_panel.csv"),
                header = TRUE)
```

Using packages

Once a package is loaded, you can use its features and functions. Here's a list of some useful and cool packages:

- `Rcmdr` - easy to use GUI
- `swirl` - an interactive learning environment for R and statistics.
- `ggplot2` - beautiful and versatile graphics (the syntax is a pain, though)
- `stargazer` - awesome latex regression and summary statistics tables
- `foreign` - reads `.dta` and other formats from inferior statistical software
- `zoo` - time series and panel data manipulation useful functions
- `data.table` - some functions to deal with huge data sets
- `sp` and `rgeos` - spatial analysis
- `multiwayvcov` and `sandwich` - clustered and robust standard errors
- `RODBC`, `RMySQL`, `RPostgresSQL`, `RSQLite` - For relational databases and using SQL in R.

Appendix - Resources

Resources

- A discussion of folder structure and data managament can be found here: https://dimewiki.worldbank.org/wiki/DataWork_Folder
- For a broader discussion of data management, go to https://dimewiki.worldbank.org/wiki/Data_Management

Git

Git is a version-control system for tracking changes in code and other text files. It is a great resource to include in your work flow.

We didn't cover it here because of time constraints, but below are some useful links, and DIME Analytics provides trainings on Git and GitHub, so keep an eye out for them.

- **DIME Analytics git page:** https://worldbank.github.io/dimeanalytics/git/
- **A Quick Introduction to Version Control with Git and GitHub:** https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004668

Appendix - More on R projects

R projects

If you want to learn more about them, we recommend starting here: https://r4ds.had.co.nz/workflow-projects.html

Appendix - Commenting

- To comment a line, write `#` as its first character

```r
# This is a comment
print("But this part is not")
```

- You can also add `#` halfway through a line to comment whatever comes after it

```r
print("This part is not a comment") # And this is a comment
```

- In Stata, you can use `/*` and `*/` to comment in the middle of a line's code. That is not possible in R: everything that comes after `#` will always be a comment

- To comment a selection of lines, press `Ctrl` + `Shift` + `C`

Appendix - Commenting

Exercise: Commenting

1. In your script panel, select all the lines of your script
2. Use the keyboard shortcut to comment these lines.
   - Shortcut: `Ctrl` + `Shift` + `C`
3. Use the keyboard shortcut to comment these lines again. What happened?