Coding for Reproducible Research

R for Stata Users

Luiza Andrade, Leonardo Viotti & Rob Marty

November-December 2018

Outline

Why are we here today?

Code is a research output

We often think of code as a mean to an end, but code is an end in itself.

Reproducibility

Can other people...

- run your code?
- recreate your results?
- understand what you are doing and why?
- make changes to your code if necessary?

Introduction

Objective

Create an R Master Script.

Content

1. Comments
2. Using R Studio to create a code index
3. Indenting
4. File paths
5. If statements
6. Using functions within functions
7. Packages
8. Loops

Introduction

- For the purpose of this training, we will assume that you are dealing with a specific folder structure
- Folder organization, though an important part of data work, is outside the scope of this course
- You can find resources about it in the appendix, and we have sent you a folder that is organized as we want it to be
- To follow today's session, go to the `DataWork/Code` folder and open the file called `Lab 4 - Coding for Reproducible Research.R`

Master scripts

- Just like in Stata, a Master script must be a map of all data work
- It should be possible to follow all data work in the data folder, from raw data to analysis output, by following the master script
- It should also be possible to run all the code for the project by simply running the Master script
- Finally, in Stata, the Master Script is where you create globals. You can do the same in R by just creating new objects

Outline

Initial Settings

A Stata do-file typically starts with a few settings:

```
clear
set maxvar 120000
set more off
```

Initial Settings

- We don't need to set the memory or the maximum number of variables in R, and the equivalent of `more` is the default in R
- However, if you saved the last RStudio session in `.Rhistory`, the objects that were in RStudio's memory last time you closed it will still be there whenever you open it again
- You can see all the objects currently in you memory in the *Environment* pane

Initial Settings

Exercise 1: Clear workspace

- Make sure the `Environment` window is open. Is anything there?
- Create an object called `foo` with any content you pick
- Type `ls()` to print the names of the object in memory
- Type `rm(foo)` to remove the `foo` object from you memory
- To remove all objects, use `rm(list=ls())`

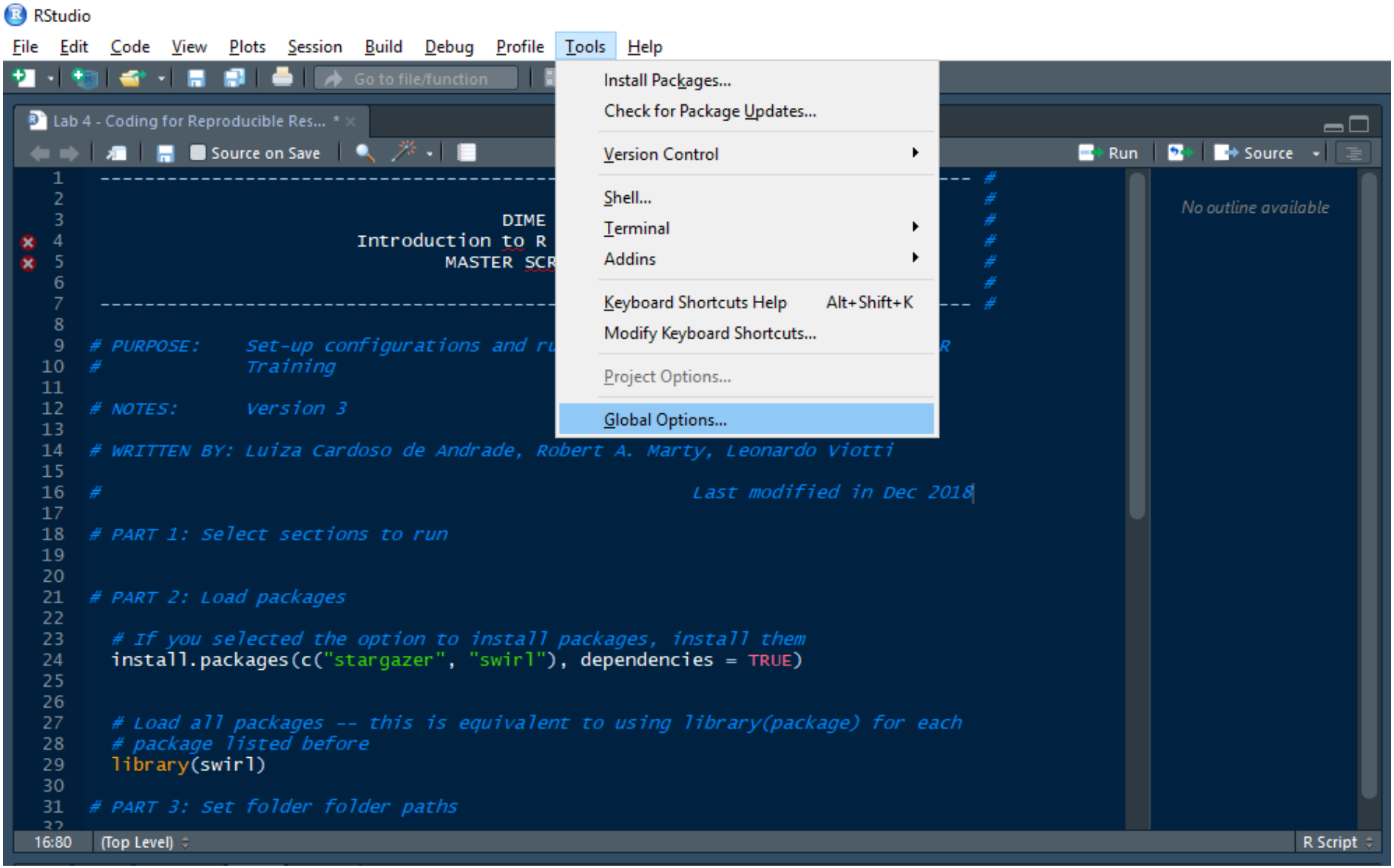
Initial Settings

Initial Settings

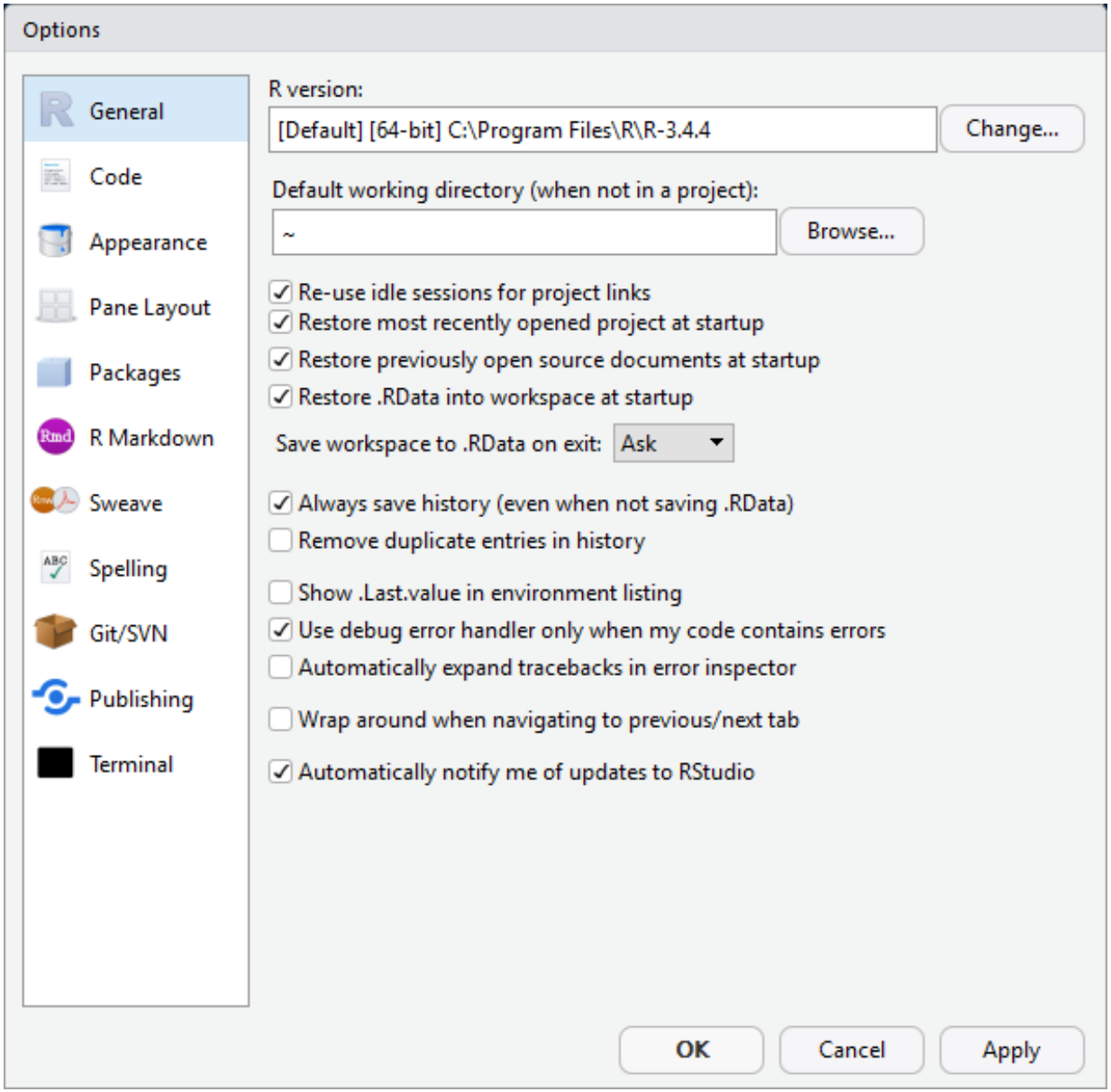
Exercise 2: No one will burn your computer

Here's how you change these settings

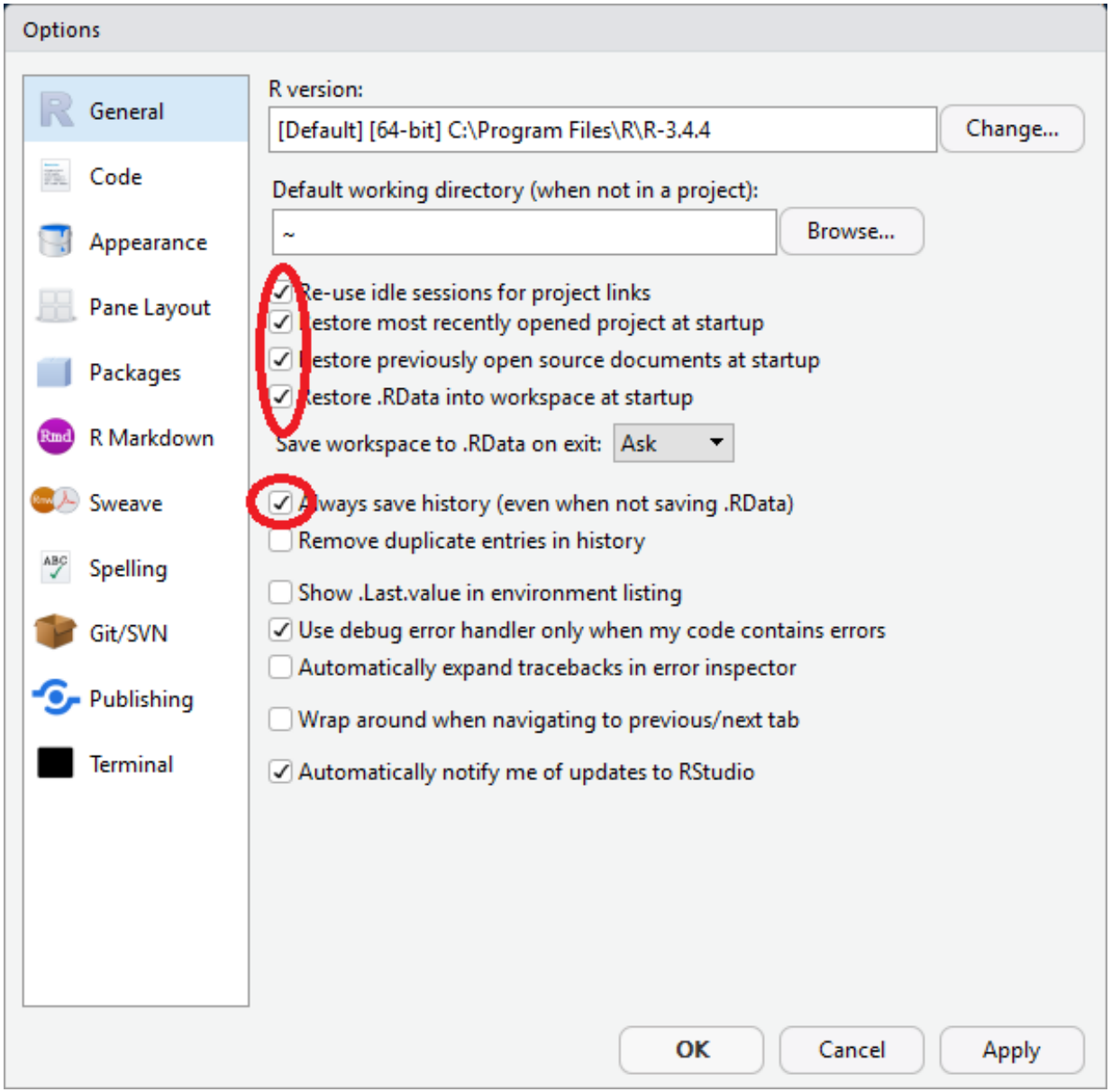
Initial Settings



Initial Settings



Initial Settings



Outline

Why comment?

Comments have two purposes:

1. Tell the reader what you are doing
2. Tell the reader why you do what you do

If you have ever read code with no comments, you probably understand why they are necessary.

Commenting

- Let's take a look at the script we just opened
- You can see that the first few lines in the script are the header, but they're not commented out
- In R, errors will not always break your code, so you should still be able to run this script
- However, not commenting out comments is still bad practice, as it makes the code harder to read

Commenting

- To comment a line, write # as its first character
- You can also add # half way through a line to comment whatever comes after it
- In Stata, you can use `/*` and `*/` to comment part of a line's code. That is not possible in R: whatever comes after # will be a comment
- To comment a selection of lines, press `Ctrl + Shift + C`

Commenting

Exercise 3: Commenting

1. Use the keyboard shortcut to comment the header of the script.
2. Use the keyboard shortcut to comment the header of the script again. What happened?

Outline

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

Exercise 4: headers

1. Open the script index and make `PART 1` a section header. Do the same for parts 2 and 3.
2. Note that once you create a section header, an arrow appears right next to it. Click on the arrows of parts 2 and 3 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

Outline

Functions inception

- In R, you can use the output of one function as the input of another, as long as they have the same format
- In fact, that's exactly what we just did when installing the packages
- To see that, select just the first argument of the `install.packages` function and press `Ctrl + Enter`
- The `c()` function, as we know, creates a vector with its arguments

```
c("stargazer", "swirl")
```

```
## [1] "stargazer" "swirl"
```

Functions inception

- The resulting vector is used as an input to the `install.packages` function
- We could also have stored this vector in the memory as an object and used that object as the input
- In fact, that's exactly what we are going to do next, so the code doesn't get too polluted as we add new packages

Functions inception

```
# Create packages object
packages <- c("stargazer",
              "swirl")

# Use it as an input to the install.packageS() function
install.packages(packages,
                 dependencies = TRUE)
```

Using packages

Exercise 6: Load packages

Load the packages you just installed, as well as the other packages used in previous sessions.

- Note that the `library` function only accepts one argument. This means that you cannot use function inception with the `packages` object here, as its output format does not match the function input. In fact, you will need to load each of them separately.

Using packages

```
library(swirl)
library(stargazer)
library(tidyverse)
library(openxlsx)
library(ggplot2)
library(plotly)
```

Outline

Looping

- If you're thinking what happens as I add more packages here, you're going in the right direction
- Repetition makes the code harder to read and to maintain

```
# The DRY rule:
DONT REPEAT YOURSELF
```

Looping

- In Stata, we'd usually use a `foreach` loop to go through a list of objects
- If you are a Stata coder, you are probably thinking of writing something like

```
* Looping through packages
foreach package in  swirl stargazer tidyverse ///
                    openxlsx ggplot2 plotly {
  library `package'
}
```

Looping

- The equivalent to that in R would be to write a for loop like this (this code won't run, so don't try it yet)

```
# A for loop in R
for (package in packages) {
    library(package)
}
```

Looping

- R, however, has a whole function family that allows users to loop through an object in a more efficient way
- 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`

Looping

- `sapply(X, FUN, ...)`: applies a function to all elements of a vector or list and returns the result in a vector. Its arguments are
  - **X:** a matrix (or data frame) the function will be applied to
  - **FUN:** the function you want to apply
  - `...`: possible function options

Looping

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

```
## [1] 1
## [1] 2
```

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

```
## [1] 1 2
```

Looping

Exercise 7: Looping in R

Use the `sapply()` function to apply the `library()` function to all packages you have selected.

- TIP: to do this without creating errors, you will need to use that `...` argument and make the `character.only` argument of the `library()` function equal to `TRUE`

Looping

```
# Load all listed packages
sapply(packages, library, character.only = TRUE)
```

Looping

A more general version is the `apply` function.

- `apply(X, MARGIN, FUN, ...)`: applies a function to all columns or rows of matrix. Its arguments are
  - **X:** a matrix (or data frame) 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
  - **`...`:** possible function options

Looping

```
# Create a matrix
matrix <- matrix(c(1, 24, 9, 6, 9, 4, 2, 74, 2),
                 nrow = 3)

# Look at the matrix
matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    6    2
## [2,]   24    9   74
## [3,]    9    4    2
```

Looping

```
    # Row means
    apply(matrix, 1, mean)
```

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

```
    # Column means
    apply(matrix, 2, mean)
```

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

Looping

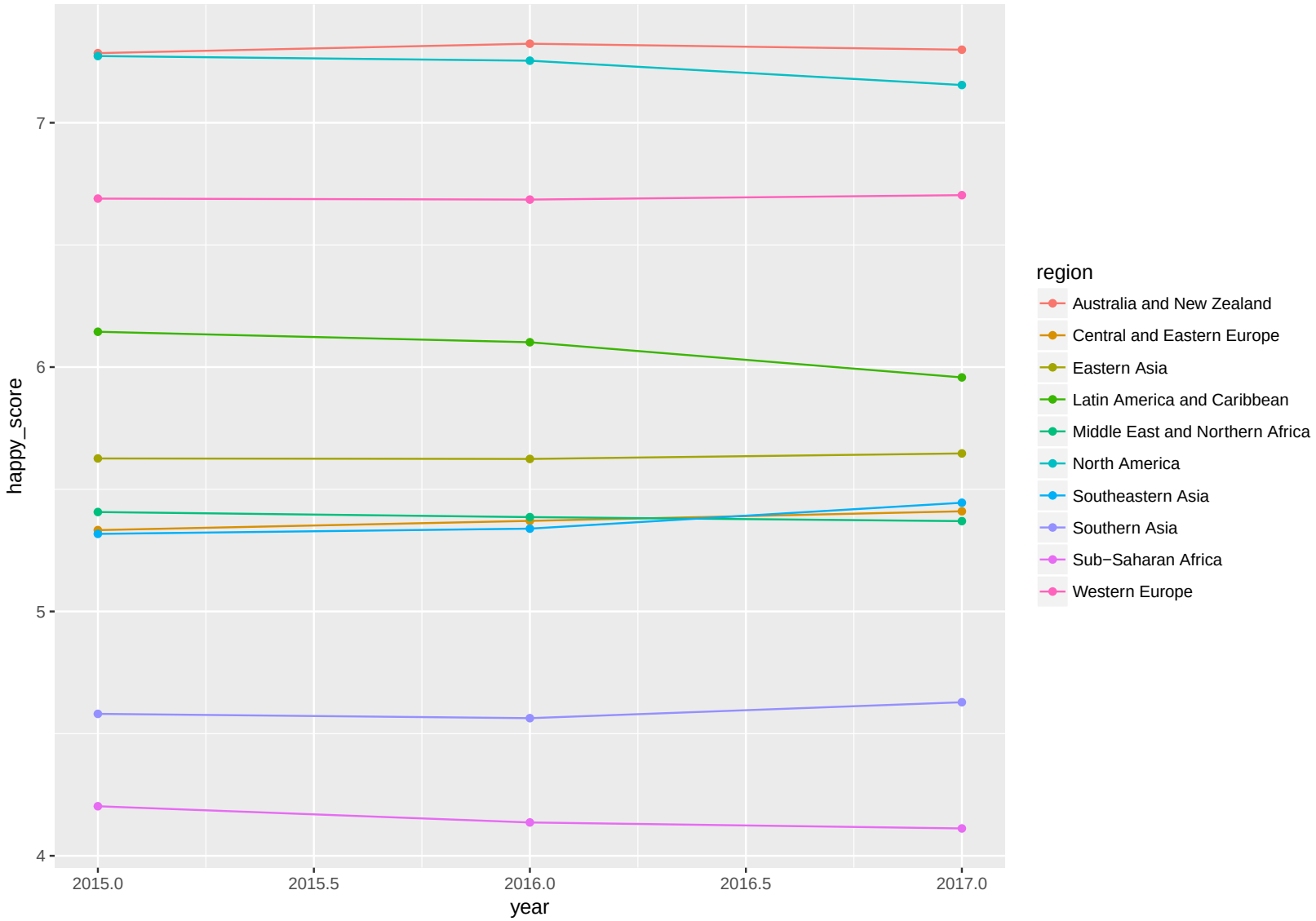
- The `apply()` function is a very powerful tool in R programming
- To explore it to its full potential, we combine it with customized functions
- This is beyond today's session, and we will come back to it in our last day, but you can give it a try on this session's assignment

Outline

Why indent?

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

Why indent?


happy_score
7
6
5
4
2015.0
2015.5
2016.0
2016.5
2017.0
year
region
Australia and New Zealand
Central and Eastern Europe
Eastern Asia
Latin America and Caribbean
Middle East and Northern Africa
North America
Southeastern Asia
Southern Asia
Sub−Saharan Africa
Western Europe

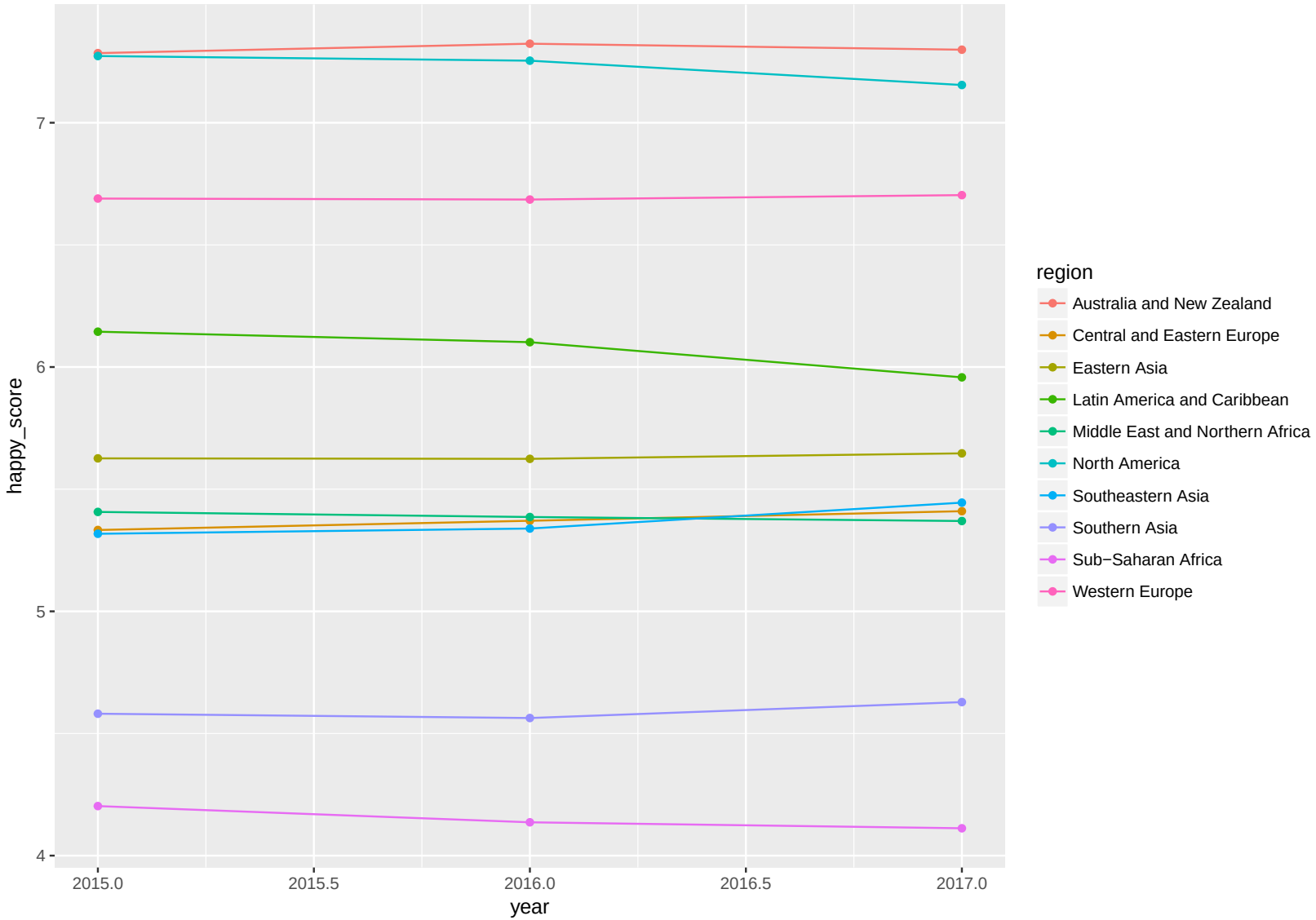
Why indent?

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

plot <-
  ggplot(annualHappy_reg,
         aes(y = happy_score,
             x = year,
             color = region,
             group = region)) +
  geom_line() +
  geom_point()

print(plot)
```

Why indent?


happy_score
7
6
5
4
2015.0
2015.5
2016.0
2016.5
2017.0
year
region
Australia and New Zealand
Central and Eastern Europe
Eastern Asia
Latin America and Caribbean
Middle East and Northern Africa
North America
Southeastern Asia
Southern Asia
Sub−Saharan Africa
Western Europe

Why indent?

- Even though R understands what unindented code says, 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

- 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

Indentation

Exercise 8: Indentation in R

To see an example of how indenting works in RStudio, 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 the round function)
2. Now add a line between the numbers in the vector.

Indentation

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

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

Outline

If statements

- Now, 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
- What can we bring from our Stata Master do-files to avoid installing packages twice?

If statements

- In Stata, section switches would be saved as locals
- In R, the equivalent to that would be to create a new object

Exercise 9: Creating an if statement

Create a dummy scalar object called PACKAGES.

- TIP: Section switches can also be Boolean objects.

If statements

- Now we need to create an if statement using this switch
- If statements in R look like this:

```
# 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 = T)
}

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

Outline

File paths

- The next important part of a Master script are file paths
- We recommend always using **explicit** and **dynamic** file paths

File paths

- Implicit and static file path:

```
# Set working directory
setwd("C:/Users/luiza/Documents/GitHub/dime-r-training/
      DataWork/DataSets/Final")

# Load data set
read.csv("whr_panel.csv",
         header = T)
```

File paths

- Explicit and static file path:

```
# Load data set
read.csv("C:/Users/luiza/Documents/GitHub/dime-r-training/
          DataWork/DataSets/Final/whr_panel.csv",
           header = T)
```

File paths

- Explicit and dynamic file path:

```
# 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 = T)
```

File paths

- File paths in R, as in Stata, are basically just strings
- Note, however, that in R we can only use forward slashes (/) to separate folder names

Exercise 10: File path to your folder

Let's start by adding the folder path to the training's folder in your computer to the beginning of `PART 3`

File paths

- You can set file paths in your master using the `file.path()` function
- This function concatenates strings using `/` as a separator to create file paths

File paths

```
# Project folder
projectFolder  <-
  "C:/Users/luiza/Documents/GitHub/dime-r-training"

# Data work folder
dataWorkFolder    <- file.path(projectFolder,"DataWork")

# Print data work folder
dataWorkFolder
```

```
## [1] "C:/Users/luiza/Documents/GitHub/dime-r-training/DataWo
```

File paths

Let's test if that worked:

Exercise 11: Test file paths

1. Save your code.
2. Start a new R session: go to `Session > New Session`. This session should be completely blank.
3. Open your Master code and turn off the switch that installs packages in your Master script
4. Add a line opening the data set in `PART 5` of your Master script

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

5. Run the whole script. If it worked, your environment should include only the `whr` data set.

Sourcing scripts

As we discussed earlier, one of the main objectives of a Master script is to be able to run all codes from a project from it.

To run an R script from another, we use a function called `source`:

- `source(file, echo, verbose)`: reads and evaluates the expressions in `file`. Its arguments are:
  - **file:** a pathname or URL to read from
  - **echo:** if TRUE, each expression is printed after parsing, before evaluation
  - **verbose:** if TRUE, more diagnostics (than just echo = TRUE) are printed during parsing and evaluation of input, including extra info for each expression

Sourcing scripts

Exercise 12: Run scripts from the master

1. Add your scripts from sessions 2 and 3 to the `Code` folder inside your `DataWork` folder.
2. Add two lines to your Master script, each one sourcing one of your session scripts
3. Run your master script.

Sourcing scripts

```
# PART 5: Run selected sections ============================
source(file.path(Code, "Lab 2"),
       verbose = T,
       echo = T)

source(file.path(Code, "Lab 3"),
       verbose = T,
       echo = T)
```

That's all, folks

- Now you have a template master script to use across this training's sessions
- Save the script that you created during this session in the *DataWork* folder. Call it *MASTER.R*
- You can also create switches for each of the sessions, and only run selected pieces of code from you Master Script

Assignment

- You can customize your loops in R by defining your own function
- This is done using a function conveniently called `function()`
- For example, if instead of just printing a number we want to print it's square, we could create a function that does both:

```
# A much more elegant for loop in R
sapply(c(1,2), function(x) x^2)
```

```
## [1] 1 4
```

Assignment

Create a function that

1. Loops through the packages listed in the `packages` vector
2. Tests if a package is already installed
3. Only installs packages that are not yet installed

- 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()
```

Outline

Using 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 the fundamental unit of shareable code
- 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 (10,000+ packages reviewed and tested) and many other online sources
- 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

Using 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("stargazer",
                 dependencies = T)
# the dependencies argument also installs all other packages
# that it may depend upon
```

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

```
library(stargazer)
```

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 dtas 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.

Using packages

Exercise 5: Installing packages

Install the `swirl` and `stargazer` packages, including packages necessary for them to run.

Using packages

```r
# Install stargazer and swirl
install.packages(c("stargazer", "swirl"), dependencies = TRUE)
```

Resources

- A discussion of folder strucutre 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

R projects

If you have used R before, you may have heard of RStudio Projects. It's RStudio suggested tool for workflow management. DIME Analytics has found that it is not the best fit for our needs, because

1. In DIME, we mainly use Stata, and we prefer to keep a similar structure in R (Stata 15 also has a projects feature, but it is not yet widely adopted)
2. We need to keep our code and data in separate folders, as we store code in GitHub and data in DropBox

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