R programming practices

R for Stata Users

DIME Analytics
The World Bank | WB Github

April 2021

Table of contents

1. Introduction
2. Initial settings
3. File paths
4. Exploring a dataset
5. Commenting
6. Creating a document outline in RStudio
7. Using packages
8. Functions inception
9. Looping
10. Custom functions
11. Indentation
12. Appendix

Introduction

Introduction

What is this session about?

- In the previous sessions, you learned how to use R in data analysis
- You are probably eager to get your hands into some data using R by now, and you would figure out **what should be in your code** for it to work
- But you would probably not know right away **how to write that**, so that in the end you might have code that is only intelligible for yourself -- and not for a very long time

Introduction

What is this session about?

- In this session, 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 code in R, and hopefully you'll be able to skip some painful steps of the "getting-your-hands-dirty" learning approach

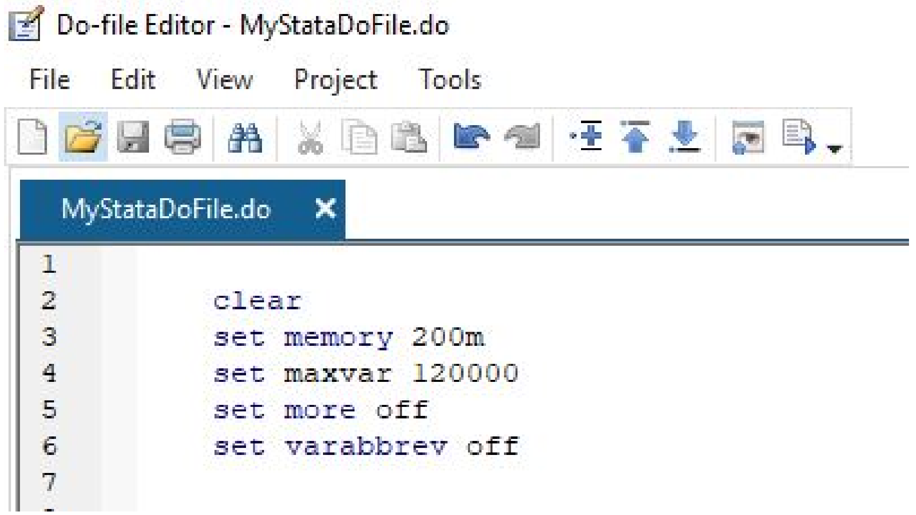
Initial settings

Initial settings

- Let's start by opening RStudio, or by reopening it if you had it opened already
- What do you see in your environment?
- If you saved the objects from the last session in `.RData`, the objects that were in RStudio's memory last time you closed it will still be there whenever you open it again
- You can also go to the `Console` panel and use the up and down keys to navigate through the commands you executed in previous sessions. They are saved by default in `.Rhistory`

Initial settings

Raise your hand if you have ever seen these lines of code before:

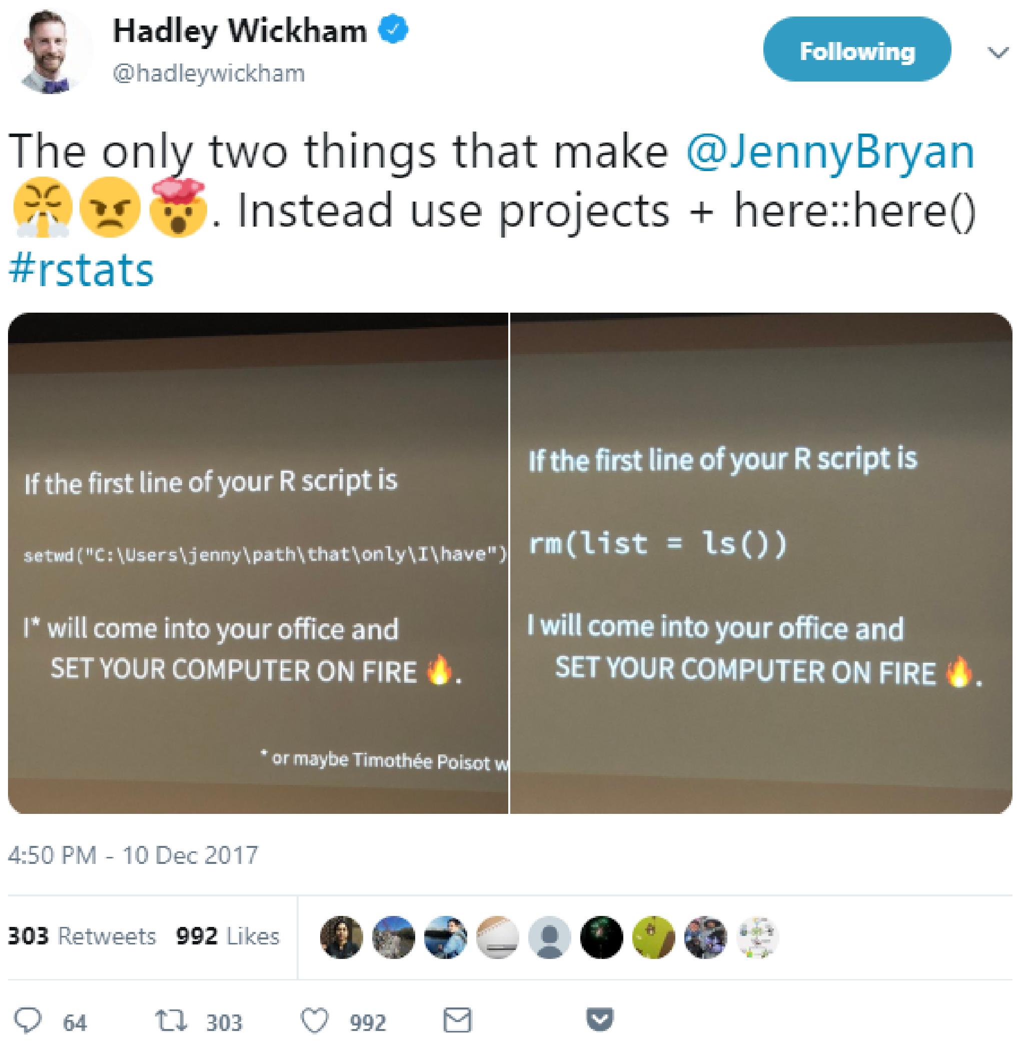

Do-file Editor - MyStataDoFile.do
File Edit View Project Tools
MyStataDoFile.do

```
clear
set memory 200m
set maxvar 120000
set more off
set varabbrev off
```

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
- You can see all the objects in your memory at any point in the `Environment` pane

Initial settings



Hadley Wickham
@hadleywickham

The only two things that make @JennyBryan 😤😠🤯. Instead use projects + here::here() #rstats

If the first line of your R script is

```
setwd("C:\Users\jenny\path\that\only\I\have")
```

I* will come into your office and SET YOUR COMPUTER ON FIRE 🔥.

* or maybe Timothée Poisot w

If the first line of your R script is

```
rm(list = ls())
```

I will come into your office and SET YOUR COMPUTER ON FIRE 🔥.

4:50 PM - 10 Dec 2017

303 Retweets 992 Likes

64 303 992

Initial settings

You don't need to use the equivalents to Stata's `cd()` and `clear all` in R if set global options correctly in RStudio.

Exercise 1: To make sure that no one will burn your computer, here's how you change these settings:

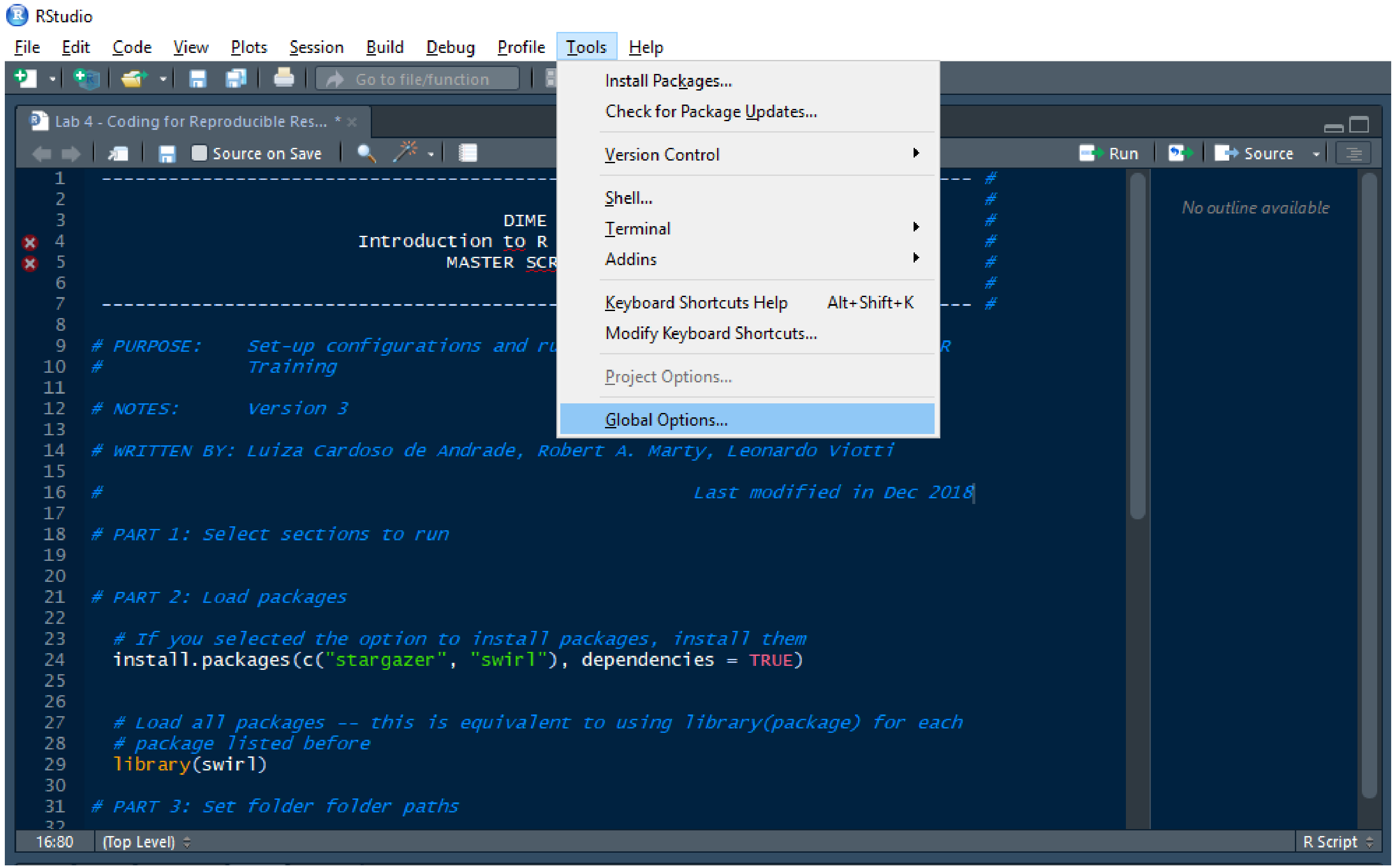
1. Go to `Tools > Global options...`

2. Make sure the following options are checked:

   - Re-use idle sessions for project links *(Note: this is only relevant for Windows)*
   - Restore most recently opened project at startup
   - Restore previously open source documents at startup
   - Restore `.RData` into workspace at startup
   - Always save history (even when not saving `.RData`)

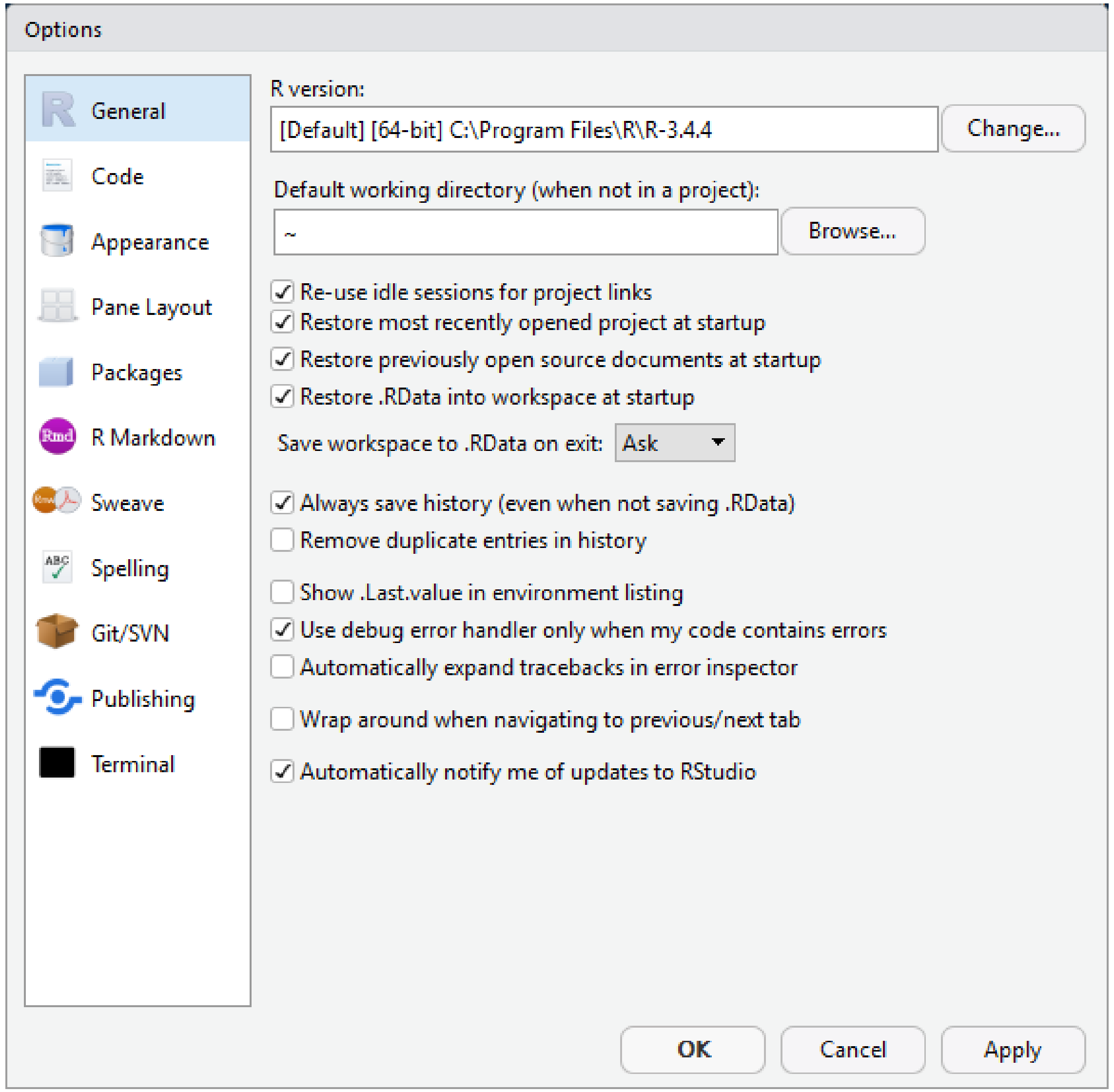
3. Now restart RStudio

Initial settings


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Lab 4 - Coding for Reproducible Res... *
Source on Save
Run
Source
Install Packages...
Check for Package Updates...
Version Control
Shell...
Terminal
Addins
Keyboard Shortcuts Help Alt+Shift+K
Modify Keyboard Shortcuts...
Project Options...
Global Options...
No outline available
16:80
(Top Level)
R Script

```
  ----------------------------------------                         #
                                                                   #
                                   DIME                            #
                         Introduction to R                         #
                                MASTER SCR                         #
                                                                   #
  ----------------------------------------                     --- #

# PURPOSE:    Set-up configurations and ru                       R
#             Training

# NOTES:      Version 3

# WRITTEN BY: Luiza Cardoso de Andrade, Robert A. Marty, Leonardo Viotti

#                                                 Last modified in Dec 2018

# PART 1: Select sections to run


# PART 2: Load packages

  # If you selected the option to install packages, install them
  install.packages(c("stargazer", "swirl"), dependencies = TRUE)


  # Load all packages -- this is equivalent to using library(package) for each
  # package listed before
  library(swirl)

# PART 3: Set folder folder paths
```

Initial settings



R version: [Default] [64-bit] C:\Program Files\R\R-3.4.4

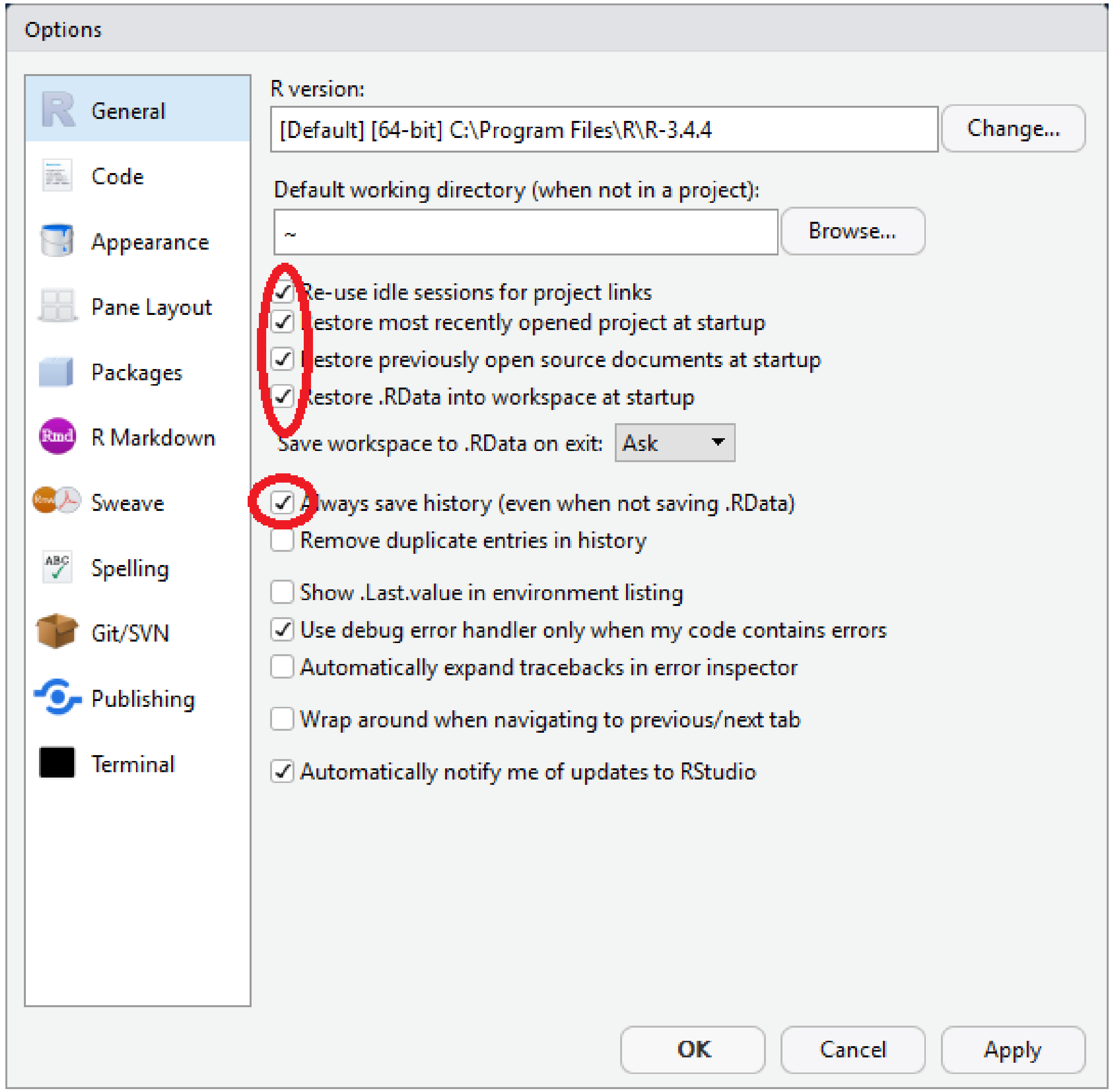
Default working directory (when not in a project): ~

- ✓ Re-use idle sessions for project links
- ✓ Restore most recently opened project at startup
- ✓ Restore previously open source documents at startup
- ✓ Restore .RData into workspace at startup

Save workspace to .RData on exit: Ask

- ✓ Always save history (even when not saving .RData)
- ☐ Remove duplicate entries in history
- ☐ Show .Last.value in environment listing
- ✓ Use debug error handler only when my code contains errors
- ☐ Automatically expand tracebacks in error inspector
- ☐ Wrap around when navigating to previous/next tab
- ✓ Automatically notify me of updates to RStudio

Initial settings



Options

R version:

[Default] [64-bit] C:\Program Files\R\R-3.4.4 Change...

Default working directory (when not in a project):

~ Browse...

- [✓] Re-use idle sessions for project links
- [✓] Restore most recently opened project at startup
- [✓] Restore previously open source documents at startup
- [✓] Restore .RData into workspace at startup

Save workspace to .RData on exit: Ask

- [✓] Always save history (even when not saving .RData)
- [ ] Remove duplicate entries in history
- [ ] Show .Last.value in environment listing
- [✓] Use debug error handler only when my code contains errors
- [ ] Automatically expand tracebacks in error inspector
- [ ] Wrap around when navigating to previous/next tab
- [✓] Automatically notify me of updates to RStudio

File paths

File paths

- For the purpose of this session, 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 shared with you a folder that is organized as we want it to be.
- For today's session:
  - Go to the OSF repository of this course (https://osf.io/b7hm6/)
  - Download the file `DataWork.zip`
  - Download and unzip the materials in a location you can remember
  - Then go to the `DataWork/Code` folder and open the file `Lab 2 - Intro II.R`
- We will use this script as a basis for the exercises, and you should modify it during this session to complete them.

File paths

- In the last session, we used the menu bar to load a data set into R
- Today, we will do that using code and referring a file's path
- 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 3: File path to your folder

Let's start by adding the folder path to the training's folder in your computer to the `projectFolder` object at the beginning of `PART 2`

```
# Add your folder path here
# Remember to replace backslashes ("\") with regular ones ("/")
projectFolder <- "YOUR/FOLDER/PATH/HERE"
```

File paths

- You can set file paths in your script using the `file.path()` function
- This function concatenates strings using `/` as a separator to create file paths
- This is similar to using globals to define folder paths in Stata

File paths

Let's test if that worked:

```r
# Project folder
projectFolder  <-
  "/home/luis_eduardo/cs_projects/dime-r-training"

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

# Print data work folder
dataWorkFolder
```

```
## [1] "/home/luis_eduardo/cs_projects/dime-r-training/DataWork"
```

File paths

Loading a dataset from CSV

Use the `read.csv()` command:

```
read.csv(file, header = FALSE, stringsAsFactors = FALSE)
```

- **file**: is the path (string) to the file you want to open, including its path, name and file extension (`.csv`).
  - Example: `/home/user/Data/whr_panel.csv`
- **header**: if `TRUE`, will read the first row as variable names (default is `header = FALSE`).
- **stringsAsFactors**: logical. See next slide for more.

File paths

Loading a dataset from CSV

- Since R 4.0.0 and beyond, `stringsAsFactors = FALSE` is the default. In every previous version, the default is `TRUE`.
- This means that if your R version is 3.X.X, R will turn any string variables into factors by default when reading a `csv` file.
- This format **saves memory**, but can be tricky if you actually want to use the variables as strings.
- You can specify the option `stringsAsFactors = FALSE` to make sure you prevent R from turning strings into factors.

File paths

Exercise 4: Test file paths

1. Save your script.
2. Start a new R session: go to `Session > New Session`. This session should be completely blank.
3. Open the code you just saved.
    - Go to `File > Open File...` and select your script
4. Add a line opening the data set in your code

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

1. Run the whole script. If it worked, your environment should now include the `whr` dataset and the path locals.

Exploring a dataset

Exploring a dataset

Some useful functions:

- `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 data set
- `str()`: general information about the structure of 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

Exploring a dataset

Exercise 5: Explore a dataset

Use some of these functions to explore the `whr` data set.

- `View()`
- `class()`
- `dim()`
- `names()`
- `str()`
- `summary()`
- `head()`
- `tail()`

Exploring a dataset

```r
# View the data set (same as clicking on it in the Environment pane)
View(whr)
```

Exploring a dataset

```r
class(whr)
```

```
## [1] "data.frame"
```

```r
dim(whr)
```

```
## [1] 470   8
```

Exploring a dataset

```r
str(whr)
```

```
## 'data.frame':    470 obs. of  8 variables:
##  $ country                 : chr  "Switzerland" "Iceland" "Denmark" "Norway" ...
##  $ region                  : chr  "Western Europe" "Western Europe" "Western Europe" "Western Europe" ...
##  $ year                    : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 ...
##  $ happiness_rank          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ happiness_score         : num  7.59 7.56 7.53 7.52 7.43 ...
##  $ economy_gdp_per_capita: num  1.4 1.3 1.33 1.46 1.33 ...
##  $ health_life_expectancy: num  0.941 0.948 0.875 0.885 0.906 ...
##  $ freedom                 : num  0.666 0.629 0.649 0.67 0.633 ...
```

Exploring a dataset

```r
summary(whr)
```

```
##    country              region                year      happiness_rank
##  Length:470         Length:470         Min.   :2015   Min.   :  1.00
##  Class :character   Class :character   1st Qu.:2015   1st Qu.: 40.00
##  Mode  :character   Mode  :character   Median :2016   Median : 79.00
##                                        Mean   :2016   Mean   : 78.83
##                                        3rd Qu.:2017   3rd Qu.:118.00
##                                        Max.   :2017   Max.   :158.00
##  happiness_score economy_gdp_per_capita health_life_expectancy    freedom
##  Min.   :2.693   Min.   :0.0000         Min.   :0.0000         Min.   :0.0000
##  1st Qu.:4.509   1st Qu.:0.6053         1st Qu.:0.4023         1st Qu.:0.2976
##  Median :5.282   Median :0.9954         Median :0.6301         Median :0.4183
##  Mean   :5.371   Mean   :0.9278         Mean   :0.5800         Mean   :0.4028
##  3rd Qu.:6.234   3rd Qu.:1.2524         3rd Qu.:0.7683         3rd Qu.:0.5169
##  Max.   :7.587   Max.   :1.8708         Max.   :1.0252         Max.   :0.6697
```

Exploring a dataset

```r
head(whr)
```

```
##       country         region year happiness_rank happiness_score
## 1 Switzerland Western Europe 2015              1           7.587
## 2     Iceland Western Europe 2015              2           7.561
## 3     Denmark Western Europe 2015              3           7.527
## 4      Norway Western Europe 2015              4           7.522
## 5      Canada  North America 2015              5           7.427
## 6     Finland Western Europe 2015              6           7.406
##   economy_gdp_per_capita health_life_expectancy freedom
## 1                1.39651                0.94143 0.66557
## 2                1.30232                0.94784 0.62877
## 3                1.32548                0.87464 0.64938
## 4                1.45900                0.88521 0.66973
## 5                1.32629                0.90563 0.63297
## 6                1.29025                0.88911 0.64169
```

Exploring a dataset

Didn't get all of those?

Don't worry, you'll see them again soon.

Commenting

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 6: Commenting

1. Go the `Lab 2 - Intro II.R` script. Select the lines under `PART 2: Set folder paths`.
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?

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 7: Headers

1. Open the script index and make `PART 1` a section header. Do the same for parts 2 and 3.
   - Remember: you create a section header by adding at least 4 trailing dashes (`-`), pound (`#`) or equal (`=`) signs in a comment line
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

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.

Using packages

Packages

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

```r
# 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**.

Using packages

Exercise 8

1. Now load the package we just installed. Use the `library()` function to do it.

Using packages

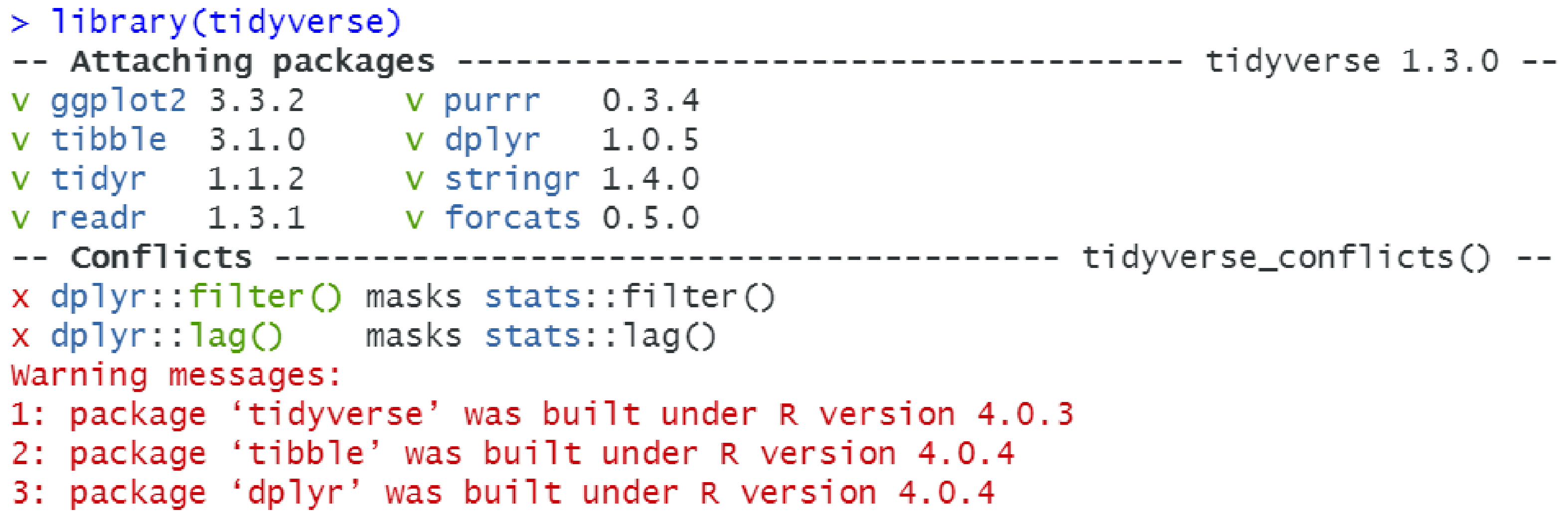
```r
library(tidyverse)
```

Notice that we used quotes around the name to install the package, but we don't need them anymore to load 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()
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
```

Using packages

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 stop the code at the lines where they occur. You can configure R to stop at warnings if you want.

Functions inception

Functions inception

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

Functions inception

```r
# Doing it the long way ----------------------------
# Create a vector with the log of the happiness score
log_score <- log(whr$happiness_score)

# Get descriptive statistics for the log vector
summary(log_score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.9907  1.5061  1.6644  1.6576  1.8300  2.0264
```

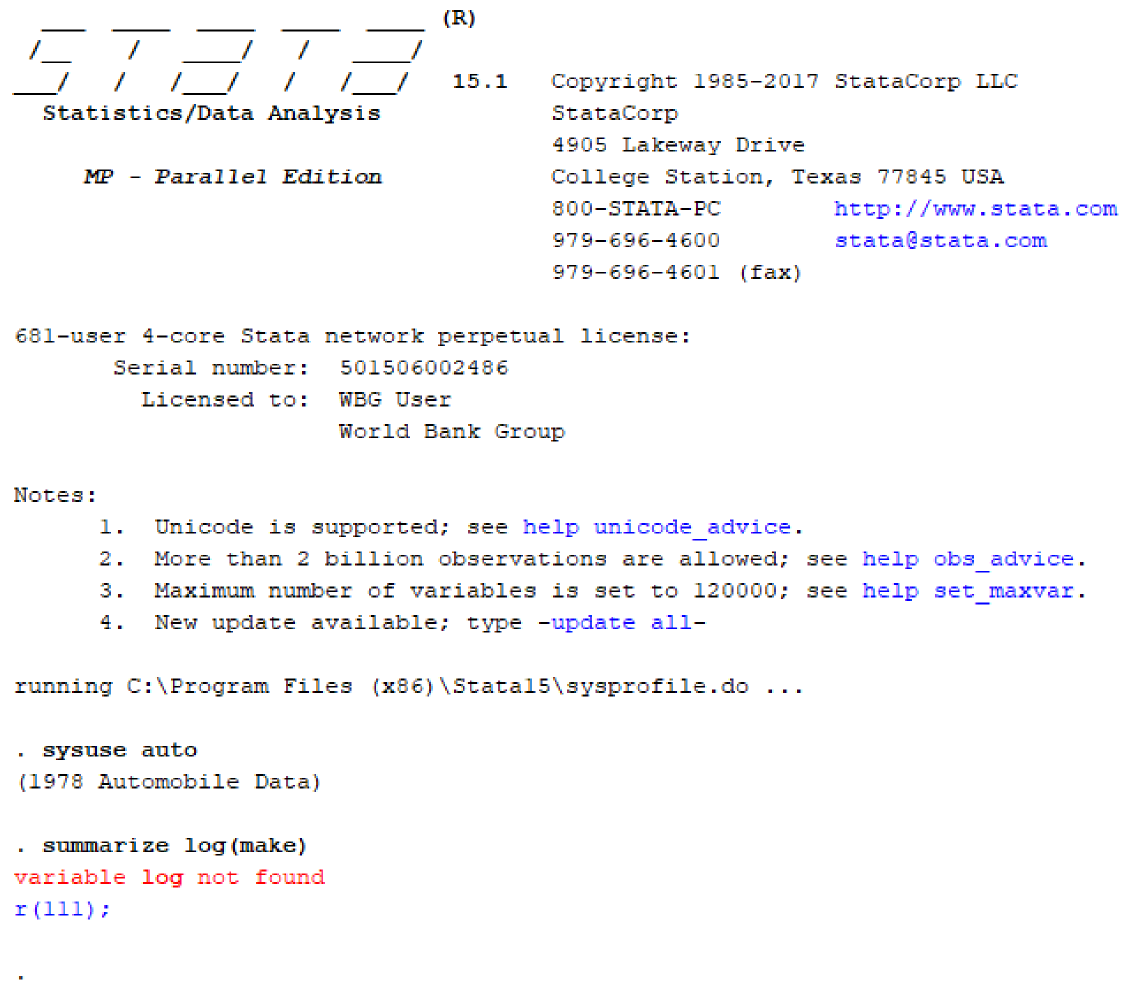
```r
# Shortcut to get to the same place ----------------
summary(log(whr$happiness_score))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.9907  1.5061  1.6644  1.6576  1.8300  2.0264
```

Functions inception

- 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

- 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
- Which is why we sometimes use pipes: `%>%`

Functions inception

```r
# 1: Doing it the long way -----------------------------
# Create a vector with the log of the happiness score
log_score <- log(whr$happy_score)

# Get descriptive statistics for the log vector
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)`

Functions inception

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

- One thing that usually gives people away as Stata users writing R code are 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

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

Looping

Loops

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Looping

Apply

- R, however, has a whole function family 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`

Looping

Apply

`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 data frame, matrix or vector the function will be applied to
- **FUN:** the function you want to apply
- **...:** possible function arguments

Looping

Loops vs Apply

```
# 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
```

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 the answers at once -- this is `apply()`

Looping

Apply

A more general version of `sapply()` is the `apply()` function.

`apply(X, MARGIN, FUN, ...)`: applies a function to all columns or rows of matrix. Its arguments are

- **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
- **... :** possible function arguments

Looping

Apply

```r
# 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

Apply

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

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

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

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

Custom functions

Custom 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

```r
square <- function(x) {

  y <- x ^ 2

  return(y)

}

square(2)
```

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

Custom functions

Exercise 9: Create a function

1. Create a function that calculates the z-score 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

Custom functions

```r
zscore <- function(x) {

  mean <- mean(x, na.rm = T)
  sd   <- sd(x, na.rm = T)
  z    <- (x - mean)/sd

  return(z)

}

summary(zscore(whr$happiness_score))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## -2.3551 -0.7579 -0.0776  0.0000  0.7590  1.9492
```

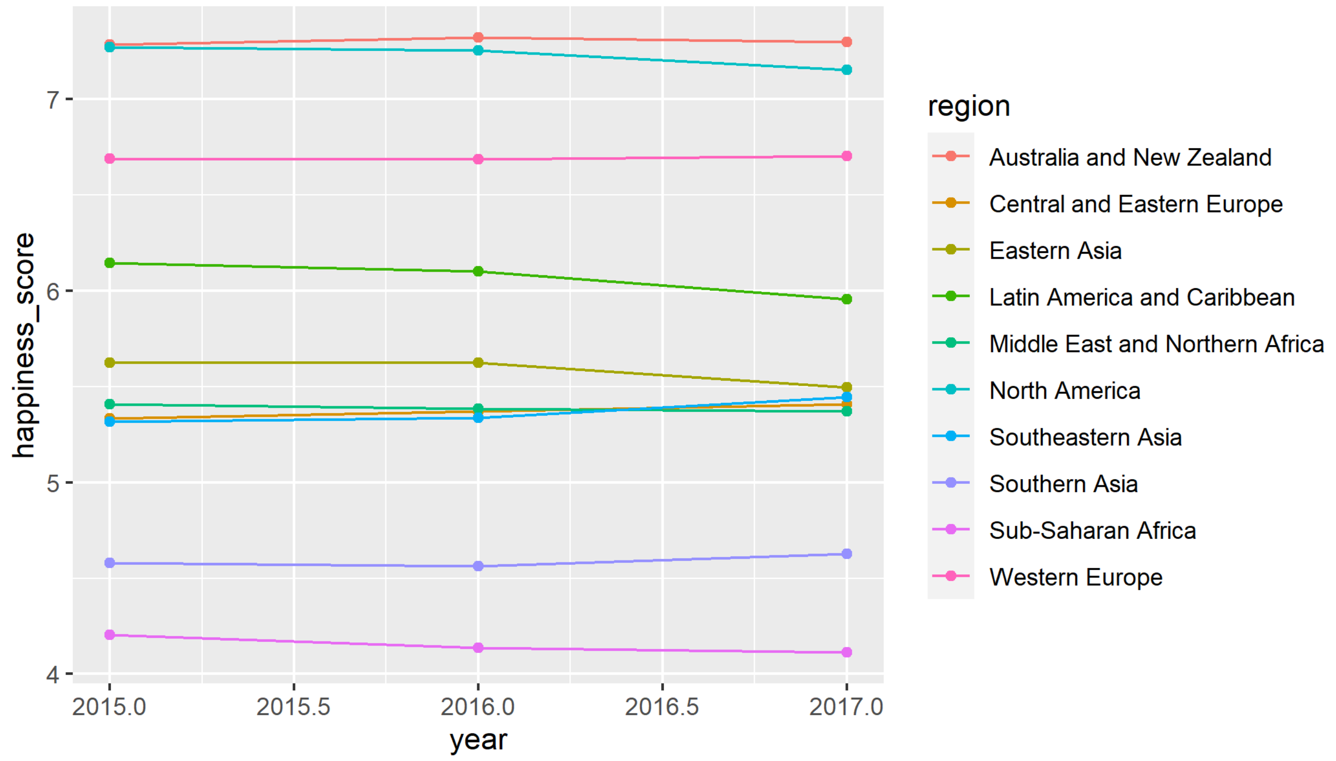
Indentation

Indentation

```
# 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)
```

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

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

Indentation



Indentation

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

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

Indentation

Exercise 8: Indentation in R

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

```r
# 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:

```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

Appendix - Assignment 1

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

Appendix - If statements

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 9: Creating an if statement

Create a dummy scalar object called PACKAGES.

- TIP: Section switches can also be Boolean objects.

Appendix - 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)
}
```

Appendix - If statements

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

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:

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

File paths best practices

- 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

File paths

- Explicit and dynamic file path:

```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

R projects

If you have used R before, you may have heard of RStudio Projects. It's RStudio's 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