7316 - INTRODUCTION TO DATA ANALYSIS WITH R

MODULE 1: GETTING USED TO R

Mickaël Buffart (mickael.buffart@hhs.se)

Table of contents

1. MODULE 1: Getting used to R

1.1 R & Statistical Programming

Unlike spreadsheet applications (like *Excel*) or point-and-click statistical analysis software (like *SPSS*), statistical programming software is based around script files, in which users write a series of commands to be performed.

R was designed as one of those. By *statistical programming*, we mean that R is a programming language designed for statistics. It has a vocabulary and grammar that is human-readable and needs to be interpreted by your computer through the R software. In this course, we will learn the basics of the R language.

Statistical programming software brings multiple benefits over point-and-click solutions:

- The data analysis process is *reproducible* and *transparent*. In R, all the commands are stored in scripts, including the data manipulations. The scripts can be stored, shared, or reused. It is the opposite of *Excel*, where it is impossible to know how a user proceeded to get a specific output.
- Due to the open-ended nature of language-based programming, R offers far more versatility in what you can do with data.
- R is an open-source language designed for statistical computing. It is the most popular choice among statisticians. It is very easy to find help and examples online, for any academic or professional purpose.
- Because of its popularity and open-source nature, R contains more pre-written functionality than any data analysis software. Currently, the CRAN repository contains more than 19000 packages: it is unlikely that you cannot find a function to perform an analysis you would wish.

1.1.1 Comparison to other statistical programming software

Stata is the traditional choice of economists. Stata is more specifically econometrics focused and is much more command-oriented. Stata has multiple technical limitations: amount of available memory; system on which it runs; you can also only work with one dataset at a time. This means that Stata is unsuited for complex data design.

- R is much better than Stata in many regards:
  - R does not bring any limitations in the amount of memory you can use or the number of data sets you can load together.
  - The range of format and data files that operates with R is almost infinite, while it is very limited with Stata (you can read Stata data files with R, but you cannot read R data files with Stata)
  - All you can do with Stata, you can do with R; the revert is untrue. In fact, you can even run Stata code within R, if needed (we will see how later during this course).

  - Stata has never been thought as a programming language: the do editor is ugly and inconvenient, and the code is complicated to write for any operations out of the ordinary. R is a programming language, and modern solution, like RStudio, contains all the tools to use it effectively as such.
  - While you can use thousands of pre-written functions, R is equally adept at programming solutions for yourself easily.

SAS is similar to Stata, but more commonly used in business & the private sector, in part because it is typically more convenient than Stata for massive dataset. While you can see it used in the books from the nineties, I do not know of anyone using SAS nowadays.

Matlab: Popular in macroeconomics and theory work, Matlab is powerful, but is much more based on programming "from scratch" using matrices and mathematical expressions.

Python: Another option based more on programming from scratch and with less pre-written commands, Python is not specific to math & statistics. It is instead a general programming language used across a wide range of fields. In the recent years, many libraries have been developed to compete with R on the statistical software side, and they have done a good job. However, Python has never been thought as a language for statistical computing, and it feels therefore less natural when used for this purpose.

The three first solutions above are proprietary software, not open-source. This means that if they decide to stop supporting a previous command or format, it will be difficult to run your own scripts in the future. They can also prevent you from using the full power of your machine (as Stata does, unless you pay more), and they are typically not available on a wide range of platform. *R* and *Python* are both open-source, free, and community driven. It is also possible to run them on a very wide range of platforms, including tablets and phones, with cloud solutions.

1.1.2 Books that are worth reading

Once you will have finished this course, you will not yet know everything you want to know about R: creating your own packages, designing interactive documents, and many other things. Bellow are some useful books to help you in your endeavor. All of them are freely available online.

- Wickham, H., & Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* O'Reilly Media, Inc. https://r4ds.had.co.nz/
  - This book is really an introduction to R. It teaches the basics of the workflow in R, data import, cleaning, types, visualization, and elementary modeling. Pretty much what we do in this class.
- Wickham, H. (2016). *ggplot2: elegant graphics for data analysis.* Springer. https://ggplot2-book.org/
  - This book is a must-read if you ever want to create beautiful graphs. It tells all you need to know about *ggplot2*, the package to generate plots with R.
- Wickham, H. (2019). *Advanced R.* CRC press. https://adv-r.hadley.nz

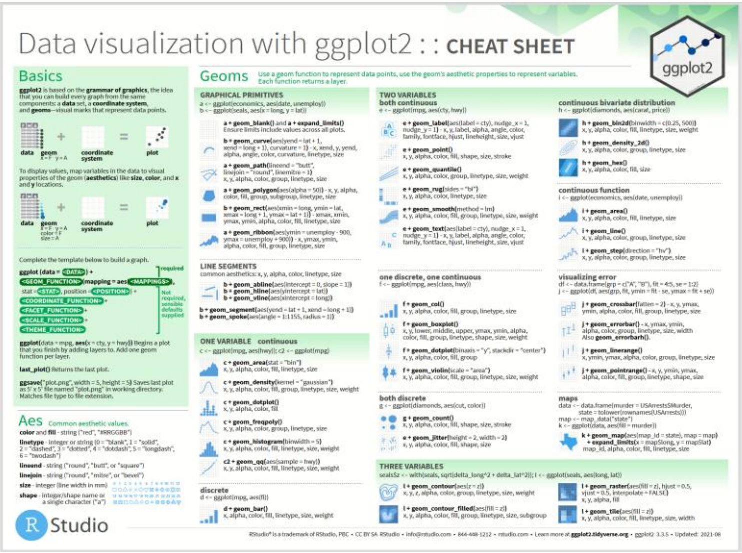
    - *Advanced R* is what comes after *R for Data Science*. It teaches about functions, objects, performance, combining R with other programming language, such as C++... This book really is about R as a programming language.
- Wickham, H. (2015). *R packages: organize, test, document, and share your code.* O'Reilly Media, Inc. https://r-pkgs.org/
    - This book will teach how to write an R package and how to share it with others.
- Xie, Y., Allaire, J. J., & Grolemund, G. (2018). *R markdown: The definitive guide.* Chapman and Hall/CRC. https://bookdown.org/yihui/rmarkdown/
    - This book will teach everything about the R markdown language: an extension of R that allows you to write books including R code and R outputs (the coursebook you are reading now has been written in Rmarkdown).

1.1.3 Useful websites for learning R

Internet is full of resources about `R`. `R` is so massively used that anything you want to do with it has likely been described and addressed somewhere on the internet. The links bellow provide the most convenient sources of information:

- StackOverflow: https://stackoverflow.com/questions/tagged/r
    - Part of the Stack Exchange network, StackOverflow is a Q&A community website for people working in programming. It is a great place to search for answers to your questions. Tons of incredibly good R users and developers interact daily on StackOverflow. The website is also very well designed: if you state a specific problem, you will likely find there an example code with a ready-made solution.
- Quick-R: https://www.statmethods.net/
    - A website with short example-driven overviews of R functionalities.
- R-Bloggers: https://www.r-bloggers.com/
    - A blog aggregator about R. R-Bloggers is a great place to learn really cool things you can do in R.
- RStudio Cheat Sheets: https://www.rstudio.com/resources/cheatsheets/
    - Very helpful 1-2-page overviews of common tasks and packages in R. The cheat sheets contain helps only for the packages and resources developed in the ecosystems of RStudio[1].

[1] RStudio and R are two different things. R is a programming language for statistical computing developed in the beginning of the nineties, and updated until today. RStudio is an Integrated Development Environment developed by RStudio PBC around 2010 by Hadley Wickham and his team. At the same time, they developed multiple convenient



Data visualization with ggplot2 : : CHEAT SHEET

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.

To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.

Complete the template below to build a graph.

```
ggplot (data = <DATA> ) +                                   required
<GEOM_FUNCTION>(mapping = aes( <MAPPINGS> ),
stat = <STAT> , position = <POSITION> ) +                   Not required,
<COORDINATE_FUNCTION> +                                     sensible
<FACET_FUNCTION> +                                          defaults
<SCALE_FUNCTION> +                                          supplied
<THEME_FUNCTION>
```

ggplot(data = mpg, **aes**(x = cty, y = hwy)) Begins a plot that you finish by adding layers to. Add one geom function per layer.

last_plot() Returns the last plot.

ggsave("plot.png", width = 5, height = 5) Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Aes Common aesthetic values.

color and **fill** - string ("red", "#RRGGBB")

linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

shape - integer/shape name or a single character ("a")

RStudio

Geoms Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

a + geom_blank() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size

a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)
x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)) - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))

b + geom_segment(aes(yend = lat + 1, xend = long + 1))
b + geom_spoke(aes(angle = 1:1155, radius = 1))

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian")
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5)
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy))
x, y, alpha, color, fill, linetype, size, weight

discrete

d <- ggplot(mpg, aes(fl))

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

both continuous

e <- ggplot(mpg, aes(cty, hwy))

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl")
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm)
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous

f <- ggplot(mpg, aes(class, hwy))

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center")
x, y, alpha, color, fill, group

f + geom_violin(scale = "area")
x, y, alpha, color, fill, group, linetype, size, weight

both discrete

g <- ggplot(diamonds, aes(cut, color))

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

e + geom_jitter(height = 2, width = 2)
x, y, alpha, color, fill, shape, size

continuous bivariate distribution

h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500))
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, fill, size

continuous function

i <- ggplot(economics, aes(date, unemploy))

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv")
x, y, alpha, color, group, linetype, size

visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j + geom_crossbar(fatten = 2) - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh()**.

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps

data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

k + geom_map(aes(map_id = state), map = map)
+ expand_limits(x = map$long, y = map$lat)
map_id, alpha, color, fill, linetype, size

THREE VARIABLES

seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l + geom_contour(aes(z = z))
x, y, z, alpha, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z))
x, y, alpha, color, fill, group, linetype, size, subgroup

l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE)
x, y, alpha, fill

l + geom_tile(aes(fill = z))
x, y, alpha, color, fill, linetype, size, width

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **ggplot2.tidyverse.org** • ggplot2 3.3.5 • Updated: 2021-08

RStudio cheatsheet example.

- DataCamp: https://www.datacamp.com/courses/free-introduction-to-r
  - DataCamp contains interactive online lessons in R.
  - Some of the courses are free (particularly community-written lessons like the one you'll do today), but for paid courses, DataCamp costs about 300 SEK / month.

1.2 Installing R and RStudio

1.2.1 R: Installation Procedure

R is an interpreted language. It means that the R code is readable by a human and an interpreter will translate the code into something readable by a computer when you try to run it. You must install an interpreter on your computer to be able to run R code.

The R interpreter is free software (GNU GPL 2) developed by the *R Core Team* and the *R Foundation for Statistical Computing*. You can download R for free from the r-project website, and install it on Windows, macOS, Linux, and other systems.

Find below direct download links to the R interpreter software:

- For Windows: https://cloud.r-project.org/bin/windows/base/R-4.2.1-win.exe
- For macOS 10.13 (High Sierra), with Intel processors:

  https://cloud.r-project.org/bin/macosx/base/R-4.2.1.pkg
- For macOS 11 (Big Sur), with Apple Arm processors:

  https://cloud.r-project.org/bin/macosx/big-sur-arm64/base/R-4.2.1-arm64.pkg

packages, and wrote books to make R convenient and accessible. It is after the work of Hadley Wickham that R became what it is today.

- For Linux, you probably want to use the version provided in your package manager. I can help you with this if needed.

The R interpreter is a command line tool: it does not offer a graphical user interface. To use R as a statistical software, as you would do with SPSS or Stata, the best option is to install RStudio.

1.2.2 RStudio: Installation Procedure

RStudio is an Integrated Development Environment (IDE) for R. You can use it to write R code and interact with the R interpreter. During the Introduction to R course, we will learn how to write and execute R code within RStudio.

The RStudio IDE is free software (AGPL 3) developed mainly by *RStudio PBC*, a public-benefit corporation, of which Hadley Wickham is the chief scientist (also him who wrote all the books on the syllabus, created ggplot2, and plenty of other tools we will discover in the class). You can download RStudio for free from the RStudio website, and install it on Windows, MacOS, Linux, and other systems.

- Download links to RStudio: https://www.rstudio.com/products/rstudio/download/#download

1.2.3 Further considerations FOR WINDOWS USERS using Dropbox or OneDrive

- If you are NOT a Windows User, *you can ignore this section*!
- If you do NOT use Dropbox or OneDrive, *you can ignore this section*!
- If you are unsure of what you are doing, no worries: you can ask me during the class and I will help you with this.

R requires libraries (also called *packages*) to run. On Windows, the packages will be stored in your `Documents` folder, in a sub-folder called `R`. If you use Dropbox or OneDrive, and you sync your `Documents` folder on the cloud, please make sure that your `R` folder is *always* available locally.

- To do just this, with *Dropbox*, if (and only if) you are syncing your `Documents` folder on Dropbox:

1. Open Windows PowerShell through the start menu
2. Enter the following command while replacing `PATH_TO_CHANGE` with the correct path to the `R/win-library` folder, usually in your `Documents` folder:

```
Set-Content -Path 'C:\Users\PATH_TO_CHANGE\Documents\R\win-library' -Stream com.dropbox.ignored -Value 1
```

3. Press Enter. After this, the `win-library` should not appear anymore on Dropbox.

- With *OneDrive*, if (and only if) you are syncing your Documents folder into OneDrive:

Unfortunately, there is no way to stop syncing a specific folder from OneDrive: it is everything or nothing. If you want to keep your `Documents` folder on OneDrive, you can always force-keeping a local copy. To do this:

1. Click on the small OneDrive cloud on your taskbar (usually, on the bottom right corner of your screen).
2. Click on Settings; go to the Settings tab.
3. Uncheck "*Save space and download files as you use them*"

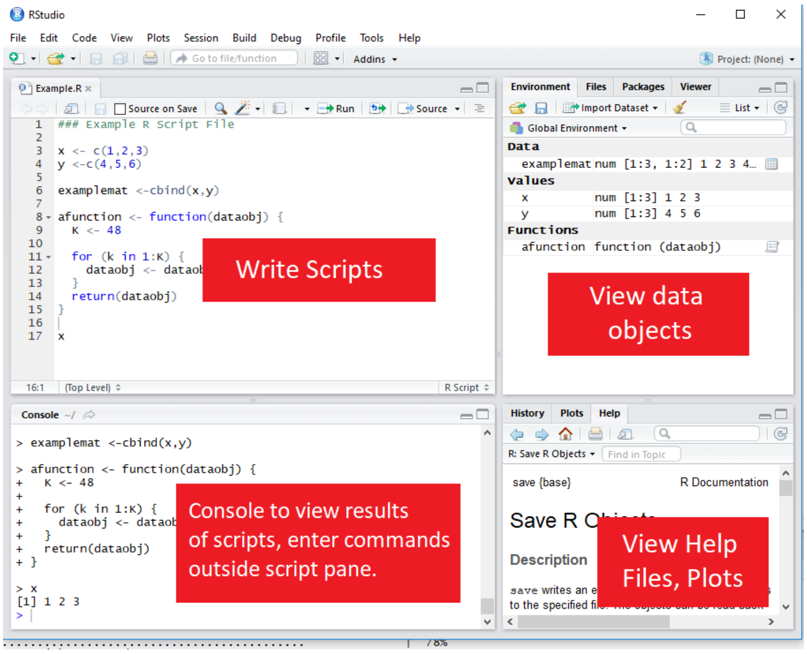
After this, your OneDrive content will be stored locally on your laptop, including the `win-library` folder.

1.3 Getting Started in RStudio

1.3.1 RStudio GUI

RStudio is an integrated development environment (IDE). This means that in addition to a script editor, it also lets you view your environments, data objects, plots, help files, *etc*.

There are four parts in the RStudio screen. They can be rearranged as it pleases you through `Tools` > `Global options` > `Pane layout`.



```
### Example R Script File

x <- c(1,2,3)
y <-c(4,5,6)

examplemat <-cbind(x,y)

afunction <- function(dataobj) {
  K <- 48

  for (k in 1:K) {
    dataobj <- dataob
  }
  return(dataobj)
}

x
```

Write Scripts

```
Data
  examplemat num [1:3, 1:2] 1 2 3 4…
Values
  x          num [1:3] 1 2 3
  y          num [1:3] 4 5 6
Functions
  afunction  function (dataobj)
```

View data objects

```
> examplemat <-cbind(x,y)

> afunction <- function(dataobj) {
+   K <- 48
+
+   for (k in 1:K) {
+     dataobj <- dataob
+   }
+   return(dataobj)
+ }

> x
[1] 1 2 3
>
```

Console to view results of scripts, enter commands outside script pane.

save {base}   R Documentation

Save R O

Description

save writes an e

to the specified fi

View Help Files, Plots

The RStudio Graphical User Interface (GUI).

1. The scripts pane is a text editor. This is where you read and write the script files. They can be `.R` files, containing R code (equivalent of `.do` files in Stata), `.Rmd` or `.qmd` files to write documents (equivalent of jupyter notebooks in Python), or files from other languages than are inter-operable with RStudio: C++, python, Stata, and many others… In general, you will work with `.R` files.
2. The console pane shows the R interpreter: there, you can run code that you do not want to save in a script, or see the outputs of the scripts you ran.
3. The two other panes contain multiple tabs that can be rearranged as you like:

- The environment tab lists all the objects that are loaded in your environment[2]: vectors, functions, dataframes, etc. (we will introduce them later in this course). In R, you can load as many objects and data frames as you want in your environment at the same time (Stata is limited to one dataframe per session). In R, the only limit is the memory of your machine.
- The Files tab is a file explorer of your working directory, convenient to find and open the script files that you want to read in the script pane.
- The Help tab shows the content of the help file for any function that you are seeking help for.
- The Plot tab shows all the plot you made since you opened your R session.
- The History tab shows all the code you ran, either through the scripts or directly in the console. This information is also saved in the .history file, in your working directory, unless you chose otherwise in the RStudio options.
- The Viewer tab shows visual outputs other than plots, such as html tables, or interactive graphs. We will use it when learning about interactive documents, in the last module.
- Finally, the Packages pane lists the *packages* (called *libraries* in Python or *macro* in Excel) loaded in the environment.

1.3.2 RStudio projects

1.3.2.1 Working in a working directory

In R, like in Stata, you interact with files (such as data files) using relative paths from a working directory. The working directory is simply the location on your computer from which R should read the scripts and data. As in Stata, you can set the path of the working directory manually, but this is not a good practice because it makes your code harder to share and reproduce: you will have to change your working directory path in all your scripts every time you move or share your work.

To avoid these problems, RStudio includes a much more powerful solution, called Projects. RStudio projects are an easy way *to work into multiple contexts, each with their own working directory, history, or source documents*. All the paths to data and source files are relative within the R projects. Then, if you share your project with someone (not only the scripts), they will not have to change anything in the code to be able to run it. You should always work within

[2] The environment contains the objects that are ready to use for your statistical purposes. As you are writing your scripts, and running your model, you may accumulate a lot of objects in your environment, some of which may have taken hours to run. In the global options, you can set it to automatically save your environment on close in an `.RData` file, and load it back where you stopped when reopening your project. If you want to empty your environment from previous objects, you can click on the small wiper in your environment tab, called *Clear objects from the workspace*. Saving or cleaning your environment does not affect your data files.

projects in RStudio. For example, this course is an R project, that you can find online on Github. We will explain what it means later in this course.

To create a New Project in RStudio:

1. Click on `File` > `New Project...`
2. If you are starting a new project from scratch, you may choose to create it in a new directory: this directory will likely contain your data, your scripts, and maybe even your output files.
3. In RStudio, you can create many different types of projects. If you wish to simply analyze some data, simply choose New Project again.
4. Then, choose a name and a location for your project. That's it! Remark that the location you choose for your project does not matter: if you decide to move the project directory somewhere else later, it will not change anything within the project, because all paths (to data and to scripts) are relative within the project directory.

Finally, to open a project (containing data, scripts, or anything else), you can either choose `File` > `Open project` in RStudio, or open the `.Rproj` file created in your project directory, from your file explorer.

Warning! Do NOT open scripts from your file explorer, always open the `.Rproj` file first. Otherwise, it is as if you were not using the project infrastructure.

If you are unsure of where your working directory is, you can use the command `getwd()` (no argument) to get the information.

1.3.2.2 Project workflow structure

If you start a big project, you will soon end up with a lot of files and folders. It is good practice to structure your working directory (*i.e.* your project) in a consistent way.

Usually, you would have something like this:

- `code/`
    - `data_prep/`
    - `analysis/`
- `data/`
    - `raw_data/`
    - `derived_data/`
- `assets/`
- `results/`
    - `tables/`
    - `figures/`

A few important good practices:

- You should store all the data in a dedicated folder (usually called `data\`). If you share your work with other people, please be careful of your right to share the data and of GDPR compliance. For example, data should NEVER be synced on Github, unless it is data that you are allowed to publicly share.
- You should always clearly separate your raw data files from the data files your derived from them (including variables that you computed yourself). That way, you avoid data loss.
- You should NEVER modify your raw data files[3]: if you modify the raw files, you may make a mistake and loose precious data forever.
- a `code/` or `script/` folder should contains all your `.R` files. It doesn't really matter how you organize them, but it is usually good practice to separate data cleaning from data analysis.
- `assets/` would usually contains the images and templates that you use to knit your documents and reports. We will learn what it means later in this course.
- Often, you would also have folders like `docs/`, `results/`, or `outputs/` to store the documents, tables, and figures that you create in your scripts (for example, with `ggplot2`, `stargazer`, and so on).

1.3.3 Executing code from the script

- To execute a section of code, highlight the code and click "Run" or use `CTRL+ENTER`.
- To execute the whole document, the hotkey is `CTRL+SHIFT+ENTER`.

1.3.4 Help files in R

You can access the help file for any given function using the help function. You can call it a few different ways:

1. In the console, use `help()`
2. In the console, use `?` immediately followed by the name of the function (no space in-between)
3. In the `Help` pane, search for the function in question.

Example:

```
# Get help on the lm (linear regression) function
?lm
```

[3] I know this is a common practice among users of Excel, Stata, and others, but this is highly discouraged; you should avoid that.

1.4 Packages in R

Packages in R are kinds of extensions. They can bring new functions or new data to the base R software, similar to user-written commands (think `ssc install`) in Stata, *libraries* in Python (think `pip install`), or *macro* in Excel. Yet, with Stata or Excel, most of the things you do probably use the core Stata commands. In R, most of the things you do are probably using packages. Once loaded in the environment, a package behave exactly as a core component of R.

There are two main sources of packages in R:

1. The most important is the CRAN repository. This is the official source of package containing a very extensive list (more than 19'000) with their documentation. To be available on CRAN, packages need to fulfill some quality criteria, checked by a team of volunteers. It does not guarantee complete security or accuracy, but at least the packages have been reviewed by someone before being available in the repository[4].
2. You can also to install packages directly from github. While github may contain more recent versions of the packages, as well as some that are not available in the official repository, those are NOT reviewed by anyone. There is therefore no guarantee at all that the package respects minimum quality criteria, or even that the package works. It might however be useful to use in some cases.

1.4.1 Installing packages from the official repository (CRAN)

- To install a package from RStudio, you can click on `Tools` > `Install packages…`, type the name of the package you would like to install, and click `Install`.
- Alternatively, you can use the function (preferably in the console) `install.packages()`
- To begin with, let's install three packages:
  - `tidyverse`, developed by Hadley Wickham (🙄): "The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures"[5].
  - `remotes`, developed by Gábor Csárdi et al.: `remotes` allows you to download and install packages from other sources than the official repository, including github.
  - `rio`, developed by Jason Becker et al.: `rio` is a package for easy data import, export (saving), and conversion.

```r
install.packages("tidyverse")
install.packages("remotes")
install.packages("rio")
```

[4] The CRAN also provides the list of authors for each package they publish: you can assess if the author is a famous unknown or someone from a serious institution.

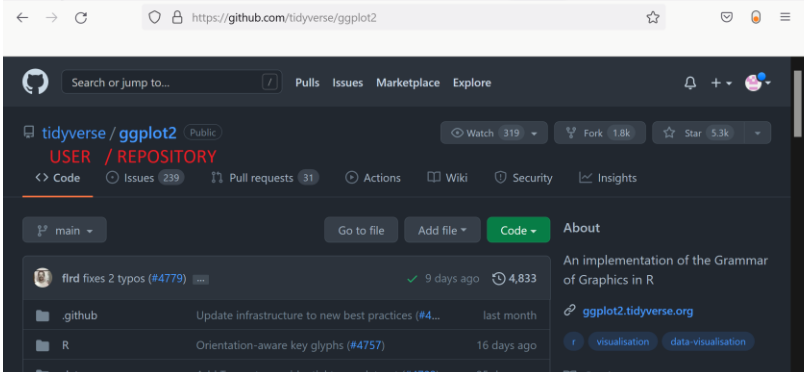
[5] Source: https://www.tidyverse.org/

1.4.2 Installing packages from Github

To install packages from github, you can use the following command:

```
remotes::install_github("user/repository")
```

where `user` is the name of the user on github who posted the package, and `repository` is the name of the package on github.



flrd fixes 2 typos (#4779) ... ✓ 9 days ago 4,833

.github Update infrastructure to new best practices (#4... last month

R Orientation-aware key glyphs (#4757) 16 days ago

An implementation of the Grammar of Graphics in R

Github package example

1.4.3 Using functions from a package in your scripts

- The best way to call a function from a package is through the following code: `package_name::function_name()`. With this, each function call is precisely related to the package it is from. This is what we used in the example above to install a package from github (with `remotes::install_github()`). Because there are thousands of different packages in R, some of them have homonym functions. Calling functions within the package namespace (i.e. using `package_name::`) avoids any confusion.

- In some cases, especially when you are using a specific package a lot in your script, it is handy to load the package once for all. To do this, use the `library(package_name)` function at the beginning of your script. If you would like to load all the functions of the `remotes` package in your environment, use:

  ```
  library(remotes)
  ```

  - Then, you can call the function in the package without mentioning the package name:

    ```
    install_github("user/repository")
    ```

1.5 Importing, exporting, selecting

1.5.1 Importing using `rio`

Previously, importing and exporting data in R was a mess, with a lot of different functions for different file formats. Stata `.dta` files alone required two functions: `read.dta` (for Stata 6-12), `read.dta13` (for Stata 13 and later), etc.

The rio package simplifies this to just one function, rio::import(), that automatically determines the file format you are trying to read and uses the appropriate function from other packages to load it. rio is able to load more than 30 different data file formats, including, csv, Excel, SAS, SPSS, Stata, Matlab, JSON, and others.

Here is an example. Let's assume you have the two following data files, from the PISA survey (2018) that I downloaded from the OECD website, placed in a data folder[6] in your project directory[7]:

- cy07_msu_sch_qqq.sas7bdat: the PISA survey dataset as an SAS data file.
- CY07_MSU_SCH_QQQ.sav: the same dataset as an SPSS data file.

Use the following command to load the data in your R environment:

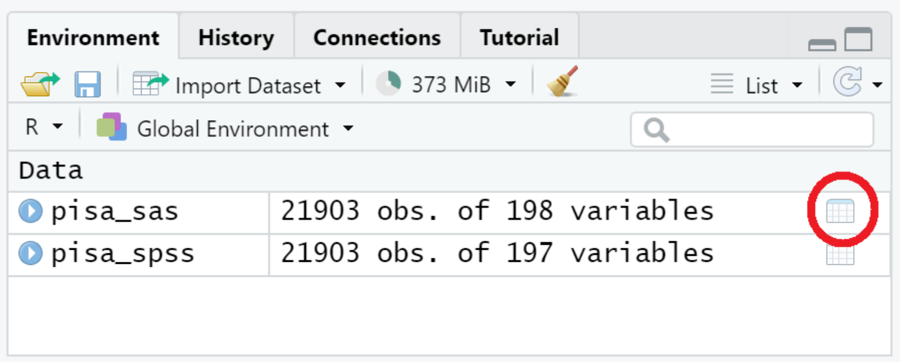
```
# Dataset in SAS format
pisa_sas <- rio::import("data/cy07_msu_sch_qqq.sas7bdat")

# Dataset in SPSS format
pisa_spss <- rio::import("data/CY07_MSU_SCH_QQQ.sav")
```

Notes:

- The arrow <- symbol indicates that you want to load the output of the function rio::import() into the objects (respectively, pisa_sas and pisa_spss)
- then, pisa_sas and pisa_spss are objects in your R environment that contains the content of the two data files. Objects can be datasets, variables, functions, strings, and many other things. In R, we always manipulate objects.
- after you ran the two commands above, you should see the two objects pisa_sas and pisa_spss, in the Environment tab of RStudio: this means that they are properly loaded in your environment. In this example, the data files are loaded as dataframe objects: these are tables containing variables. You can visualize the content of the objects by clicking on the small table at the end of the corresponding line in your Environment tab.

[6] In practice, you may name the data folder as you like, but it is common practice to name it data.

[7] Do not forget: before starting any new data work, create a new R project in a new directory. The directory will contain your data, scripts, and outputs.



Data

pisa_sas	21903 obs. of 198 variables
pisa_spss	21903 obs. of 197 variables

Visualize a dataset in RStudio

- the first part `data/`, is not part of the filenames: it is the folder where your data files are located in your project directory.

If you are interested in a dataset that is part of a package, you can load it by simply calling its name. For example, to use the `Wages1` dataset stored in the package `Ecdat`, you call:

```
wages <- Ecdat::Wages1
```

1.5.2 Display the structure of the dataset

After loading your dataset, you may want to see its structure. You can see the structure in the environment tab, by clicking on the small arrow before the `dataframe` name. Alternatively, you can use the function `str()`. It will display the names of the variables within your `dataframe`, their `types`, and a few first observations.

```
str(pisa_sas)
```

1.5.3 Exporting data

If you want to save an object, *e.g.* `pisa_sas`, into a new file, you can use:

```
# SOLUTION 1:
saveRDS(pisa_sas, "data/pisa_sas.Rds")

# SOLUTION 2:
rio::export(pisa_sas, "data/pisa_sas.Rds")
```

The function `saveRDS()` is the base function to save an R data object into a file. `rio::export()` is the wrapper from the `rio` package. They both lead to the exact same result.

`.Rds` is the standard format to save data in R. I advise you to use it, because you are sure that the data are saved exactly as you see them in R (no loss of information). However, if you need to export data into a file compatible with other software (such as Excel or Stata), you can do it with `rio::export()`.

Warning: Some format will result in a loss of data, for example, when the data types you use are not compatible with the chosen file format.

```
# Saving the data into an Excel sheet
rio::export(pisa_sas, "data/pisa_sas.xlsx")
```

```
# Saving the data for Stata
rio::export(pisa_sas, "data/pisa_sas.dta")
```

1.6 Using version control with R

Above, I introduced projects and explained that all your scripts should be part of *RStudio projects*. Using projects, you can manage the complete workflow of your analyses over time, from data access and cleaning to the generation of documents to share. You can also save a RStudio environment that is unique to the project (the `.RData` file): that allows you to close RStudio and come back to it at any time, to find everything exactly as you left it[8].

But there is more! With project, you can use version control: this is the equivalent of *track change* in Word documents, with the possibility to share your project with collaborators.

1.6.1 What is version control?

Version control is a way to track changes in files. The most popular software for version control is Git. Git was authored by *Linus Torvalds* (also the creator of Linux) because he wanted to collaborate with other programmers without talking to them. Of course, this may sound a bit dull, but it was a tremendous success, because it saves a lot of time. For any project using Git: you can track, approve, or revert any change of any users in a smooth and transparent way. Version control allows you to:

- see a history of every change made to files
- annotate changes
- revert files to previous versions
- track user changes on a file

1.6.1.1 Git and Github

With Git, you can manage versions of your files on your local computer. Git is usually combined with cloud solution, to track changes on multiple computers, or with multiple users.

One famous cloud solution for Git is Github. Github is a commercial cloud service owned by Microsoft. You can easily create a free Github account, with some limitations, but enough resources to manage a personal R project[9].

Important Note: Github is a great tool to share and manage R codes with peers, but it is, to my knowledge, not GDPR compliant (the E.U. regulation regarding data protection). This

[8] Maybe it does not sound like much, but you can't do that with Stata…

[9] Many alternatives, equally compatible with RStudio, exist, such as BitBucket, Gitlab, or Amazon Web Service. It is also quite easy to configure your own Git server, if you do not want to use proprietary solutions. Yet Github is certainly the most popular, and the most ready-to-use platform at the moment.

means that you CANNOT store data on Github. This course also covers how to use Github without compromising your data.

1.6.1.2 Some vocabulary

- Repository: to simplify, the *repository* is your working directory. With Git, contrary to Word, you do not track changes on a single file, but on the content of a folder (*e.g.* your working directory). This folder is called a repository. With Github, the repository has a *local* copy (on your computer) and an *online* copy (on the cloud), just as Dropbox or OneDrive.
- Clone: With Github, everything starts with a repository on the cloud. First thing you will need to do is to create a local copy of your repository (you cannot really make direct changes on the online copy). This is called: to *clone* the repository.
- Commit: In the Git language, *committing* means that you want to record a change that you made on a file. This is like making a snapshot of the current version of a file. Any committed change can be restored to any previous commit.
- Push and pull: Once you committed changes on a file, you can *push* them on github, or *pull* someone else's commits in your local repository.
- Merge: When multiple members of a team modify the same code without knowing about the other one's changes, it results in two version of the same files. The version can then be *merged* into one, or kept in separate *branches*.
- Branch: a *branch* is a different version of the same repository that has not yet been merged with the *main branch*.

1.6.2 Setting up the tools

1.6.2.1 Installing Git for local version control

In this course, we will not cover how to use Git for local version control, nor how to configure Git for use directly through RStudio. The reason is that it requires a bit more configuration efforts than what we want to spend in his course: we therefore privilege a ready-to-use solution that you can use in a few minutes. If you ever dreamed of using Git and Github within RStudio but never dared to ask, you can find further information about it in the great book of Jennifer Bryan: *Happy Git with R*. Otherwise, bellow, I provide you with a simpler solution.

1.6.2.2 Setting up Github for remote version control

In general, you use version control to be able to share work and progress with a team, or with a greater audience. One easy way to do it is to use Github. GitHub connects an online repository to the local Git repository on your computer, "pushing" and "pulling" changes between the local and remote repositories. To get ready with Github:

1. Join github here: https://github.com/signup. When asked, it is good practice to use `firstnamelastname` as username, if available (mine is `mickaelbuffart`), but it does not really matter. Your username is unique. This is the way to find you on github.
2. Download and install Github Desktop. You can install Github Desktop on Windows and MacOS.

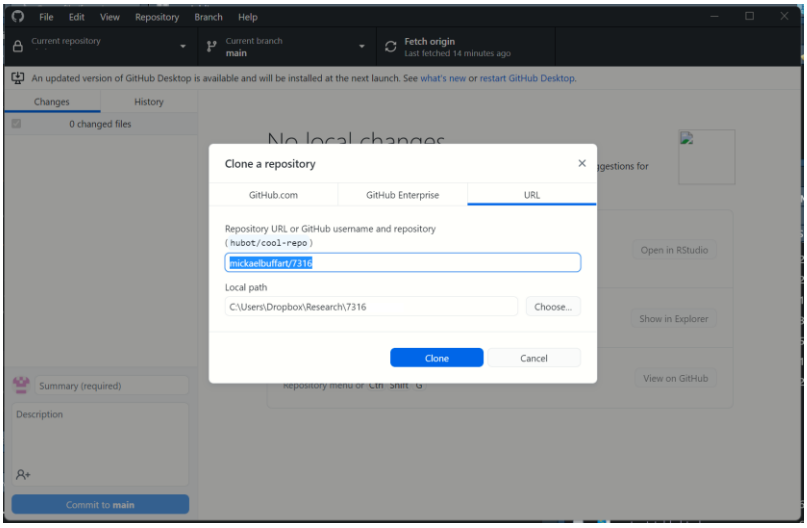
Github desktop is the easiest client to interact with Github. It is able to track changes in a repository just as Git does it, and it is able to *clone* repositories, and *push* and *pull commits* securely between your local copy and Github. Also, it has an easy interface, where you need no command line tool to commit any changes.

1.6.3 Using Github and Github Desktop

1.6.3.1 Cloning a repository

Once you installed github Desktop, you can start cloning repositories on your computer. For example, this course is on github. Let's try to clone it. To clone a repository:

1. Open Github Desktop (and log in with the account you created, above)
2. Click on `File` > `Clone repository...`
3. There, you can enter the URL of the repository you want to clone.
    a. For example, if you want to clone this course on your computer, click on URL, and enter `mickaelbuffart/7316` .
    b. In the local path, choose where you want to clone your repository on your computer.



An updated version of GitHub Desktop is available and will be installed at the next launch. See what's new or restart GitHub Desktop.

Repository URL or GitHub username and repository (`hubot/cool-repo`): mickaelbuffart/7316

Local path: C:\Users\Dropbox\Research\7316

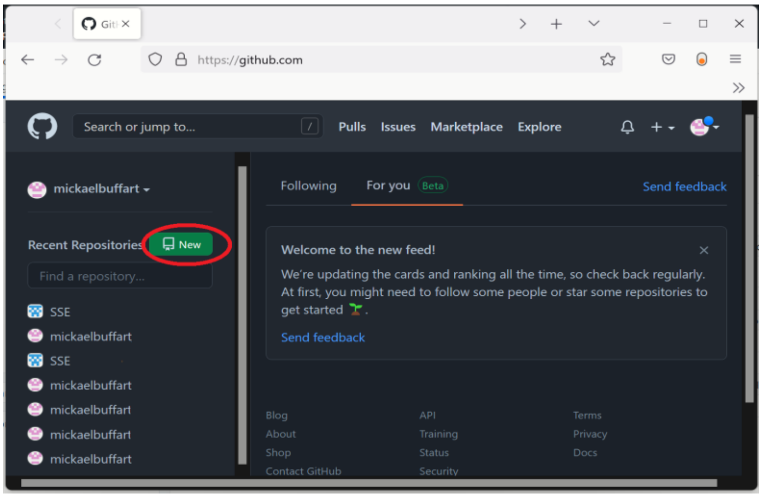
Clone a repository

- Now, you can find the local copy of your repository in the local path you chose.
- You can also start copy or editing files in your local repository. Note: If you clone a public repository (such as this course), you will not be able to push commits in it, unless its owners grant you rights to do so.

1.6.3.2 Setting up a new repository

If you start a new project, you need to create a new repository on Github. The easiest way is to:

1. Log in your github account
2. Click on New



Welcome to the new feed!

We're updating the cards and ranking all the time, so check back regularly. At first, you might need to follow some people or star some repositories to get started 🌱.

Send feedback

Create a new repository

3. Supply GitHub with a repository name (*i.e.* this is the name of your working directory; it cannot contain space or special characters)

4. Choose whether or not the repository should be public (*i.e.* ANYONE can see your repository) or private[10] (*i.e.* only the people you explicitly allow can see your repository).

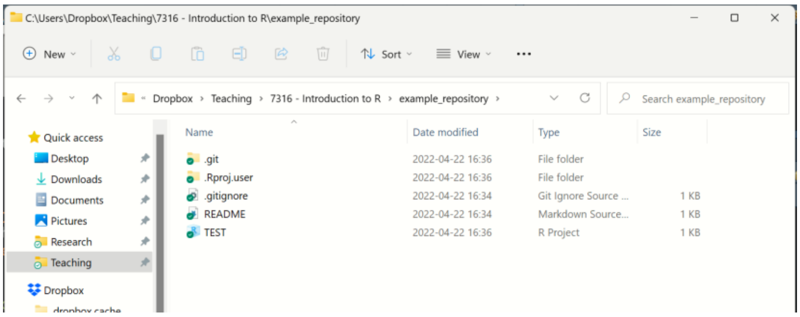
5. In *Add .gitignore*, choose R (if you are creating an R project).

6. You can also choose to add a license or a readme file[11], as you see fit.

That's it! You can now clone a local copy of your repository, and start populate it with files.

1.6.3.3 Managing `.gitignore`

After you created an RStudio project in your local repository, you should see the following files:


C:\Users\Dropbox\Teaching\7316 - Introduction to R\example_repository
New
Sort
View
Dropbox › Teaching › 7316 - Introduction to R › example_repository ›
Search example_repository
Quick access
Desktop
Downloads
Documents
Pictures
Research
Teaching
Dropbox
dropbox cache

Name	Date modified	Type	Size
.git	2022-04-22 16:36	File folder	
.Rproj.user	2022-04-22 16:36	File folder	
.gitignore	2022-04-22 16:34	Git Ignore Source ...	1 KB
README	2022-04-22 16:34	Markdown Source...	1 KB
TEST	2022-04-22 16:36	R Project	1 KB

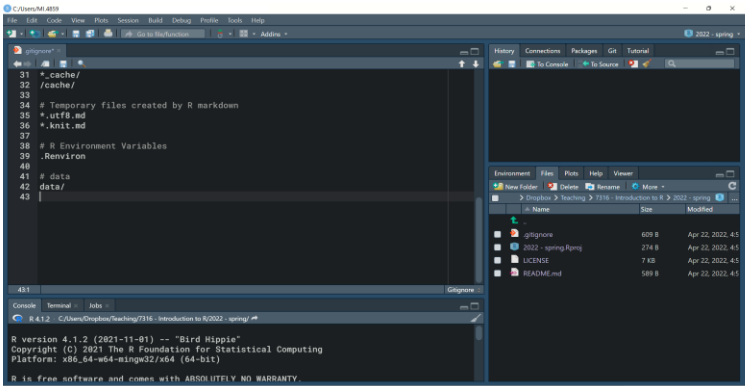
Content of your local repository

[10] With a free Github account, you cannot share a private repository; if you want to collaborate with other people without making your repository public and without a paid subscription, you can do it by creating a team. It has many limitations compared to a paid account, but that would likely do the job for a school project with two or three partners.

[11] I strongly advise you to add both, especially if you want to share your work with other people at some point, but you can also edit this later.

Note that `.git` and `.R.Rproj.user` are hidden and may not appear on your screen. In any case, you should NEVER modify the content of those folder, or you might loose control of your changes.

- Your repository contains a file named `.gitignore`. This is a text file listing all the files you want to ignore from the version control (this means that the files and folders listed here will NOT be committed in the version control, and NOT be pushed to Github).
- To make sure your data do not end up somewhere on the internet, it is a good practice to make git ignore your `data/` folder. To add your data folder to `.gitignore`, you can:
    1. Open `.gitignore` in RStudio or with a text editor
    2. add `data/` on a new line at the end of the file
    3. Save
- You can also do that with any other file or folder in your local repository on which you do not want to track changes.



```
*_cache/
/cache/

# Temporary files created by R markdown
*.utf8.md
*.knit.md

# R Environment Variables
.Renviron

# data
data/

```

```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
```

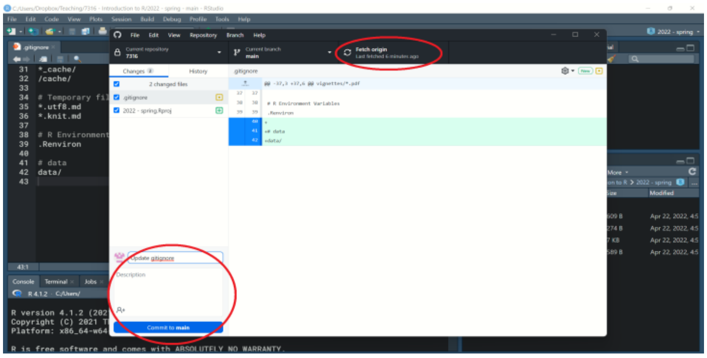
Name	Size	Modified
.gitignore	609 B	Apr 22, 2022, 4:5
2022 - spring.Rproj	274 B	Apr 22, 2022, 4:5
LICENSE	7 KB	Apr 22, 2022, 4:5
README.md	589 B	Apr 22, 2022, 4:5

C:/Users/MI.4859
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
2022 - spring
.gitignore
History Connections Packages Git Tutorial
To Console To Source
Environment Files Plots Help Viewer
New Folder Delete Rename More
Dropbox > Teaching > 7316 - Introduction to R > 2022 - spring
43:1
Gitignore
Console Terminal Jobs
R 4.1.2 · C:/Users/Dropbox/Teaching/7316 - Introduction to R/2022 - spring/

The .gitignore file.

1.6.3.4 Committing changes in Github Desktop

Once you made a few changes in your repository (like, editing the `.gitignore` file), you can open Github Desktop again, and you will see that your current repository list the files you have change.



Current repository: 7316
Current branch: main
Fetch origin — Last fetched 6 minutes ago

☑ 2 changed files
☑ .gitignore
☑ 2022 - spring.Rproj

```
@@ -37,3 +37,6 @@ vignettes/*.pdf

# R Environment Variables
.Renviron
+
+# data
+data/
```

Update gitignore

Description

C:/Users/Dropbox/Teaching/7316 - Introduction to R/2022 - spring - main - RStudio
File Edit View Repository Branch Help
Changes 2
History
.gitignore
Commit to main

Commit a change

- On the down-left corner, after you wrote a description of your changes, you can commit them. All the changes you commit will permanently appear in the history tab.

1.6.3.5 Syncing changes with a remote repository

If you are just tracking changes with a local repository, commit is sufficient to manage version control, but if you are using version control with an remote (*i.e.* online) repository, you will need sync your commit and the commits of others with github. For this, you have one button, that I highlighted in red in the screenshot above: `fetch origin`. This button has 3 states:

1. `fetch origin` checks if any changes occurred online since the last time you checked.
2. `Push origin`: to push your local commits to Github.
3. `Pull origin`: to pull commits from others (on Github) in your local repository.

After pushing your commit, you can see them on Github.

With git, the changes made by different members of the team are asynchronous (*i.e.* you do not need to be connected to internet to make local changes, but you will have to push them to the cloud later on). It results that you may, from time to time, manage version conflicts, if multiple users modified the same chunk of code (*i.e.* lines of code) at the same time. Git helps you with that.

1.6.3.6 Viewing previous commits

- To view previous versions of the files (along with annotations supplied with the commit message), click on the `History` button.
- From there, you can see not only a *difference* view of the file changes, but you can also open the document exactly how it was written in a previous commit.
- If you want to revert changes, you *could* explicitly revert the file from the contextual menu (*revert changes in commit*), or simply copy over the file with code from the previous commit.

1.7 Writing code in R

Very soon, your scripts in R will be hundreds of lines. To make sure everyone can decipher your code easily, you should always comply to some basic style rules. To know more about them, you can read the styleguide of Hadley Wickham (him again…). Bellow, I list some minimum good practices you should comply to:

1.7.1 Write short lines of code

Unlike Stata, in R, you do not need any special command to write code on multiple lines: it is already the default (functions are written with parentheses and brackets, so it is clear when a command ends). Therefore, there is no excuse for long lines. Accepted styleguides suggest a 80-characters limit for your lines.

RStudio has the option to show a ruler for 80-character margins. Use it!

1. Go to `Tools` > `Global Options` > `Code` > `Display`

2. Select Show Margin and enter 80 characters.

1.7.2 Give air to your code!

It may seem useless when you start, but having a properly designed code increase readability, and reduce the risk of mistakes. Do not hesitate to break lines and to space instructions.

- GOOD practice:

```
tmp <- c(2, 5, 3, 7, 8, 10, 1, 156)

tmp <- tmp * 2

summary(tmp)
```

- BAD practice:

```
tmp<-c(2,5,3,7,8,10,1,156)
tmp<-tmp*2;summary(tmp)
```

Technically, both chunks are equally readable by the R interpreter, but when you will have thousands of lines of codes (it comes faster as you think…), one style will give you headaches; the other one won't.

1.7.3 Use comments to explain what you are doing in your code

To further improve the readability of your code, use comments everywhere. A comment is a text that will NOT be interpreted by the R interpreter. This allows you to write anything you like in your code. To create a comment in R, use a hash (#). For example:

```
# Here I sum 2 with 2
2 + 2

[1] 4
```

You can comment or uncomment multiple lines by highlighting them and pressing `CTRL+SHIFT+C`.

1.7.4 Naming conventions

R does not have constraints for naming objects (*e.g.* variables). However, if you want your dataset to be compatible with other software, such as Stata, you may want to follow more restrictive naming conventions. To make sure that your variable names could be read in most statistical software, you should:

1. use only lowercase
2. use only letters, integers, and _ in your variable names (avoid spaces, dots, hyphens, comas, accents, or any special characters)
3. avoid starting a variable name with an integer (you can however use integers later in the name)
4. prefer short names as much as possible

Again, this does not matter for R, but you may run into errors when interacting with Stata if you do not follow those simple rules.