MODULE 1: GETTING USED TO R

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

1. R & Statistical Programming

Unlike spreadsheet applications (like *Excel*) or point-and-click statistical analysis software (like *SPSS*), statistical programming software is based on 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 human-readable vocabulary and grammar 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***. All the commands are stored in scripts, including the data manipulations in R. 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 easy to find help and examples online for academic or professional purposes.
- R contains more pre-written functionality because of its popularity and open-source nature than any data analysis software. Currently, the CRAN repository contains more than 19'000 packages: it is unlikely that you cannot find a function to perform the desired analysis.

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, such as the amount of available memory, the system on which it runs, and the number of datasets you can open 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 limit 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 minimal 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. You can even run Stata code within R if needed (we will see how later during this course).
  - Stata is not designed 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 solutions, like RStudio, contain all the tools to use it effectively.
  - 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, partly because it is typically more convenient than Stata for massive datasets. While you can see it used in the books from the nineties, I do not know 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 fewer 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 recent years, many libraries have been developed to compete with R on the statistical software side, and they have done an excellent job. However, Python has never been thought of as a language for statistical computing, and it feels less natural when used for this purpose.

The three first solutions above are proprietary software, not open-source. This means that running your scripts in the future will be difficult if they stop supporting a previous command or format. 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 platforms. *R* and *Python* are both open-source, free, and community-driven. It is also possible to run them on a wide range of media with cloud solutions, including tablets and phones.

1.2 Books that are worth reading

Once you 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. Below are some valuable books to help you in your endeavor. All of them are freely available online.

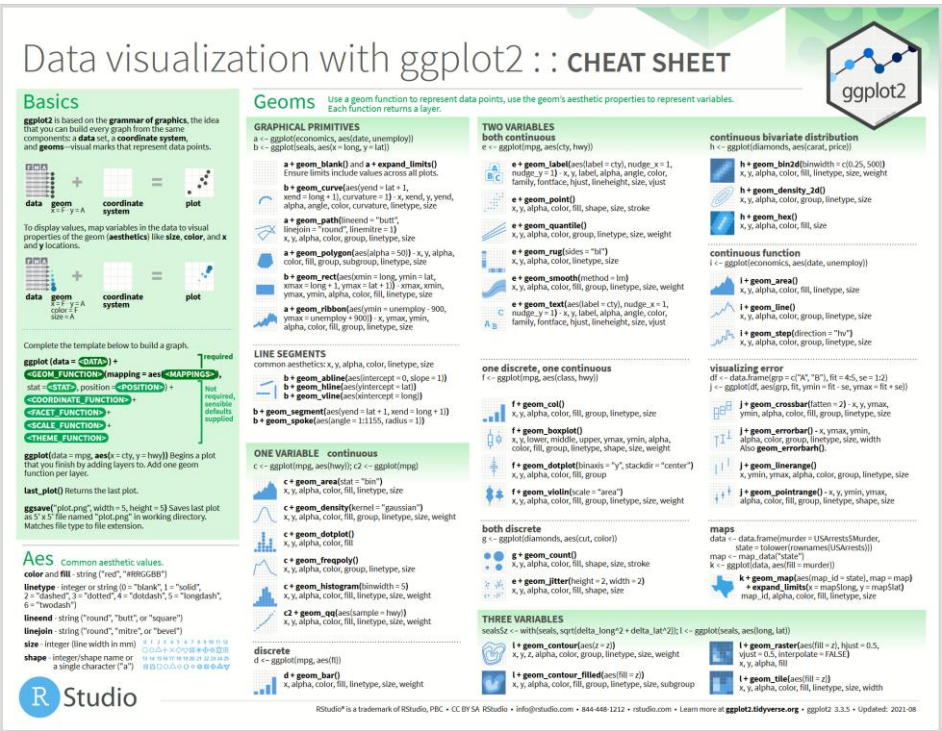
- Wickham, H., & Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* O'Reilly Media, Inc. https://r4ds.had.co.nz/
  - This book is an introduction to R. It teaches the workflow basics in R, data import, cleaning, types, visualization, and elementary modeling. That is 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, and combining R with other programming languages, such as C++... This book is about R as a programming language.

1.3 Useful websites for learning R

The 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 elsewhere on the Internet. The links below provide the most convenient sources of information:

- **RStudio Cheat Sheets:** https://www.rstudio.com/resources/cheatsheets/
  - Very helpful 1-2-page overviews of everyday tasks and packages in R. The cheat sheets contain help 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 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.



RStudio cheatsheet example.

- **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 excellent 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 an example code with a ready-made solution.
- **DataCamp:** https://www.datacamp.com/courses/free-introduction-to-r
  - DataCamp contains interactive online lessons in R.
  - Some courses are free (particularly community-written lessons like the one you'll do today), but DataCamp costs about 300 SEK / month for paid courses.
- **ChatGPT**: https://chat.openai.com/auth/login
  - ChatGPT is not a website to learn R, but it can help generate example code when you get stuck. ChatGPT works well if you formulate small and precise requests. For complex requests, the code generated will likely contain mistakes with the current version of ChatGPT.
  - For this course, I encourage you to avoid using ChatGPT and always think of solutions yourself, as a learning process. Otherwise, you will never master `R`.

2. Installing R and RStudio

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 it into something readable by a computer when you try to run it. You must install an interpreter on your computer 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://cran.r-project.org/bin/windows/base/R-4.3.1-win.exe
- For macOS 11 (**Big Sur**), with Apple Arm processors:

  https://cran.r-project.org/bin/macosx/big-sur-arm64/base/R-4.3.1-arm64.pkg
- You probably want to use the version provided in your package manager for Linux. 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**.

2.2 RStudio: Installation Procedure

RStudio is an Integrated Development Environment (IDE) for R, proper 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.

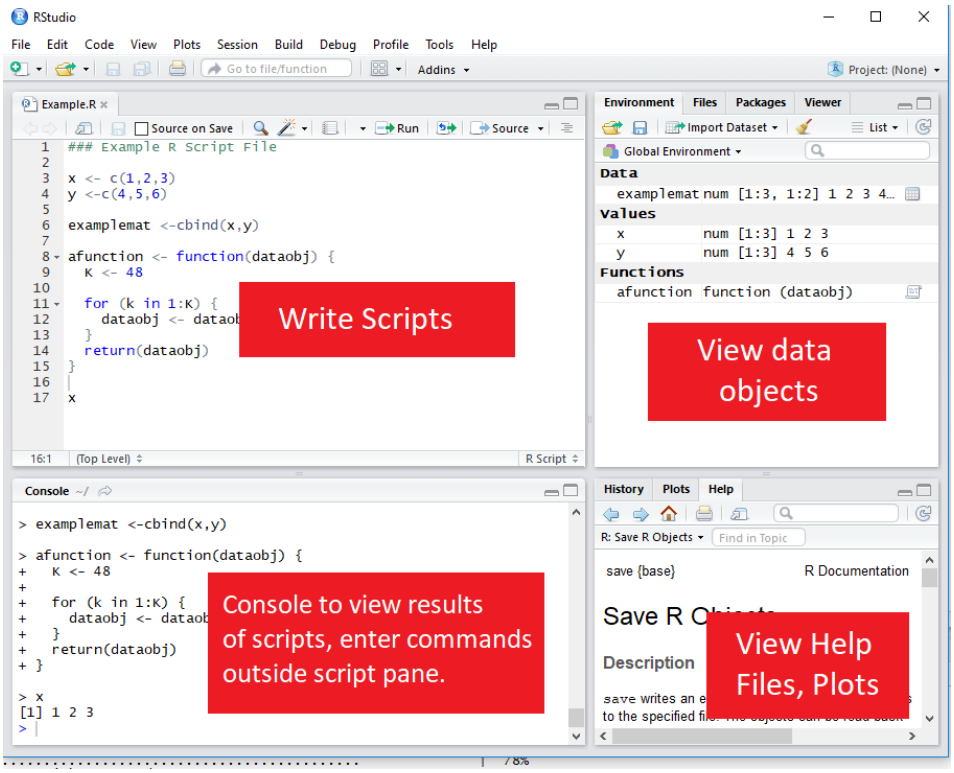
- Download links to RStudio: https://posit.co/download/rstudio-desktop/#download

3. Getting Started in RStudio

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`.



The RStudio Graphical User Interface (GUI).

1. The **script** pane is a text editor. This is where you read and write the script files. They can be `.R` files containing R code (the equivalent of `.do` files in Stata), `.qmd` files to write documents (this is the equivalent of Jupyter notebooks in Python), or files from other languages compatible with RStudio: C++, Python, Stata, and many others... In general, you will work with `.R` and `.qmd` files.
2. The **console** pane shows the R interpreter, where you see the outputs of the scripts you ran. You can also type code to run directly in the console.
3. The two other panes contain multiple tabs that can be rearranged as you like:
   - The **environment** tab lists all the objects loaded in your environment[2]: vectors, functions, dataframes, etc. (we will introduce them later in this course). In R, you can simultaneously load as many objects and data frames as you want in your environment (Stata is limited to one dataframe per session). In R, the only limit is the memory of your machine.

[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.

- The **Files** tab is a file explorer of your working directory, convenient to find and open the script files you want to read in the script pane.
- The **Help** tab shows the help file's content for any function you seek help for.
- The **Plot** tab shows all the plots you have made since you opened your R session.
- The **History** tab shows all the code you ran through the scripts or in the console. This information is also saved in the `.history` file in your working directory unless you choose otherwise in the RStudio options.
- The **Viewer** tab shows visual outputs other than plots, such as HTML tables or interactive graphs. In the last module, we will use it when learning about interactive documents.
- Finally, the **Packages** pane lists the *packages* (called *libraries* in Python or *macro* in Excel) loaded in the environment.

3.2 RStudio projects

3.2.1 Working in a working directory

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

To avoid these problems, RStudio includes a much more robust solution called **Projects**. RStudio projects are an easy way to work in multiple contexts, each with its 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 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 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 analyze some data, 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: moving the project directory somewhere else later will not change anything within the project because all paths (to data and scripts) are relative within the project directory.

Finally, to open a project (containing data, scripts, or anything else), you can 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.

3.2.2 Project workflow structure

If you start a big project, you will soon have many files and folders. It is good practice to consistently structure your working directory (*i.e.*, your project).

Usually, you would have something like this:

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

A few essential good practices:

- You should store all the data in a dedicated folder (usually called `data\`). If you share your work with others, please be careful of your right to share the data and GDPR compliance. For example, data should **NEVER** be synced on GitHub unless it is data that you are allowed to share publicly.
- You should always clearly separate your raw data files from the data files you derived from them (including variables that you computed yourself). That way, you avoid data loss.
- You should **NEVER EVER EVER** modify your raw data files[3]: if you modify the raw files, you may lose precious data forever. Always save your changes into new data files.
- A `code/` or `script/` folder should contain all your `.R` files. It doesn't matter how you organize them, but separating data preparation from data analysis is usually good practice.

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

- `assets/` would usually contain the images and templates you use to knit your documents and reports. We will learn what it means later in this course.
- You would often also have folders like `docs/`, `results/`, or `outputs/` to store the documents, tables, and figures you create in your scripts (for example, with `ggplot2`, `stargazer`, and so on).

3.3 Executing code from the script

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

3.4 Help files in R

You can access the help file for any 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
```

4. Packages in R

Packages in R are kinds of extensions. They can bring new functions or data to the base R software, similar to user-written commands (think `ssc install`) in Stata, *libraries* in Python (think `pip install`), or *macros* 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 behaves precisely as a core component of R.

There are two primary sources of packages in R:

1. The most important is the CRAN repository. This is the official source of packages containing a pervasive list (more than 19'000) with their documentation. To be available on CRAN, packages must fulfill some quality criteria, checked by a team of volunteers. It does not guarantee complete security or accuracy, but someone has at least reviewed the packages before being available in the repository[4].

[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.

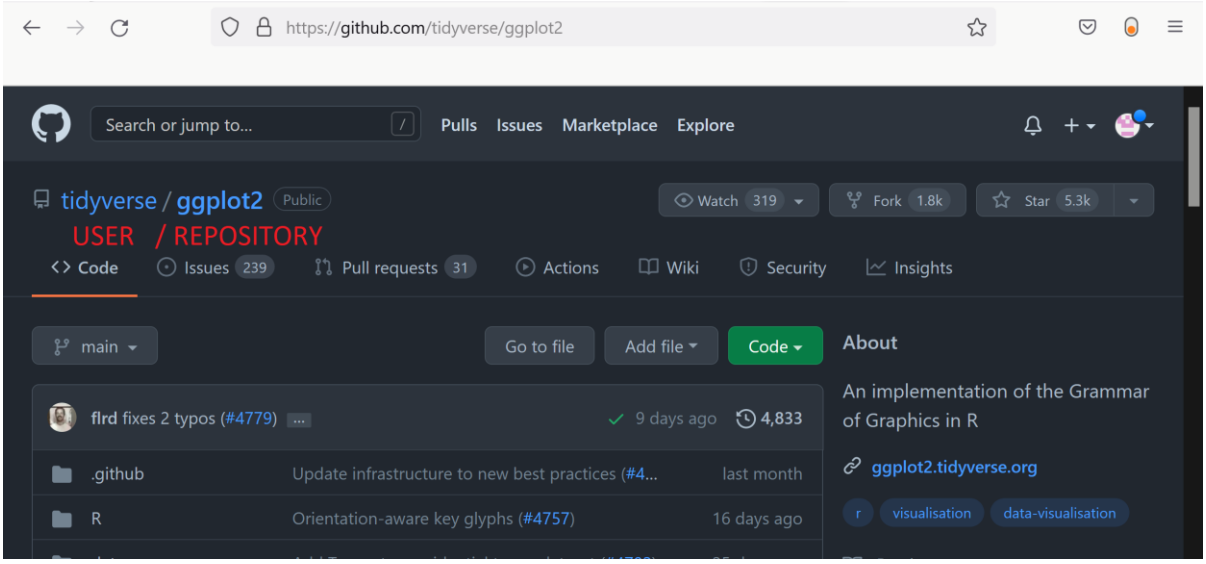
2. You can also install packages directly from GitHub. While GitHub may contain more recent versions of the packages and some that are not available in the official repository, those are **NOT reviewed by anyone**. Therefore, there is no guarantee that the package respects minimum quality criteria or even that the package works. It might, however, be helpful to use in some cases.

4.1 Installing packages from the official repository (CRAN)

- To install a package from RStudio, click Tools > Install packages..., type the package name you want, 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` allow you to download and install packages from sources other 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.

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

4.2 Installing packages from GitHub



GitHub package example

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

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.

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. We used this in the example above to install a package from GitHub (with `remotes::install_github()`). Because there are thousands of packages in R, some have homonym functions. Calling functions within the package namespace (i.e., using `package_name::`) avoids 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 the following:

```
library(remotes)
```

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

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

5. Importing, exporting, selecting

5.1 Importing using `rio`

Previously, importing and exporting data in R was a mess, with many 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` can 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 and placed in a `data` folder[6] in your project directory[7]:

- `cy07_msu_sch_qqq.sas7bdat`: the PISA survey dataset as a SAS data file.

[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.

- `CY07_MSU_SCH_QQQ.sav`: the same dataset as an SPSS data file.

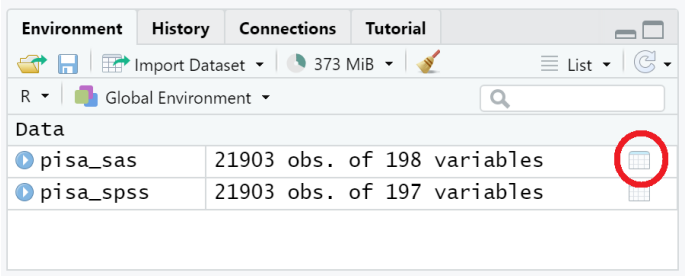
Use the following command to load the data in your R environment:

```
# 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.
- The first part, `data/`, is not part of the filenames: it is the folder where your data files are located in your project directory.
- After you run 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 loaded correctly 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.



Visualize a dataset in RStudio

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
```

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 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)
```

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 same result.
- The first argument is the name of the object you save. The second argument is the path where you save it on your computer. The two arguments are separated with `,`.

`.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 formats will result in a loss of data, for example, when the data types you use are incompatible 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")
```

Finally, instead of saving data files, you can save a complete R environment (an `.RData` file). Saving the environment allows you to close RStudio and return to it anytime to find everything exactly as you left it[8]: variables, functions, dataframes, and objects, with no loss.

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.

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

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. It was a tremendous success because it saved much time. For any project using Git: you can track, approve, or revert any change of any users smoothly and transparently. Version control allows you to:

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

- See a history of every change made to files.
- Annotate changes.
- Revert files to previous versions.
- Track user changes on a file.

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

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 like Dropbox or OneDrive.
- **Clone:** With Github, everything starts with a repository on the cloud. First, you must create a local copy of your repository (you cannot make direct changes to the online copy). This is called: to *clone* the repository.
- **Commit:** In the Git language, *committing* means you want to record a change 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 commit changes on a file, you can *push* them on GitHub or *pull* someone else's commits in your local repository.
- **Merge:** When multiple team members modify the same code without knowing about the other one's changes, it results in two versions of the same files. The version can then be *merged* into one or kept in separate *branches*.

[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.

- **Branch:** a *branch* is a different version of the same repository that has not yet been merged with the *main branch*.

6.2 Setting up the tools

6.2.1 Installing Git for local version control

In this course, we will not cover how to use Git for local version control or configure Git for use directly through RStudio. It requires more configuration effort than we want to spend in his course. Therefore, we privilege a ready-to-use solution you can use in a few minutes. If you have 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, below, I provide you with a simpler solution.

6.2.2 Setting up Github for remote version control

In general, you use version control to share work and progress with a team or 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 a username, if available (mine is `mickaelbuffart`), but it does not 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.

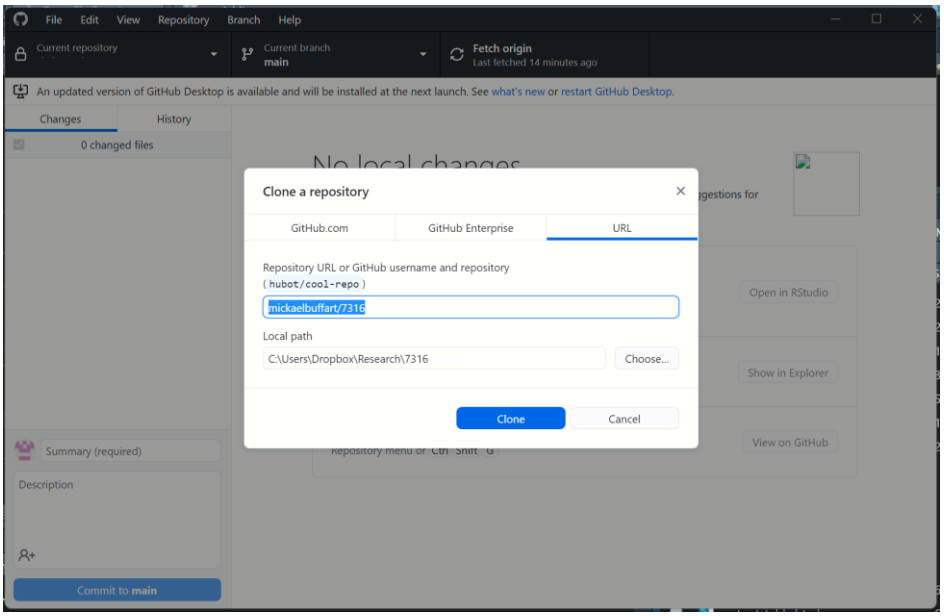
Github desktop is the most accessible client to interact with Github. It can track changes in a repository just as Git does, *clone* repositories, and *push* and *pull commits* securely between your local copy and Github. Also, it has an easy interface, where you need no command line tool to commit any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

6.3 Using Github and Github Desktop

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 repository URL 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.



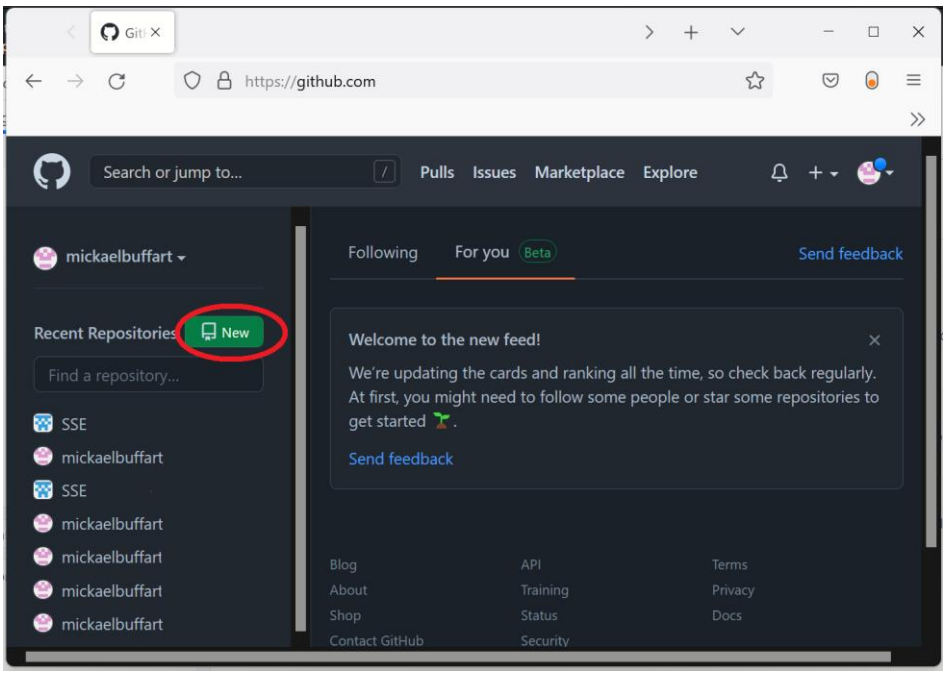
Clone a repository

Now, you can find the local copy of your repository in the local path you chose. You can also start copying or editing files in your local repository. **Note:** If you clone a public repository (such as this course), you cannot push commits in it unless its owners grant you rights.

6.3.2 Setting up a new repository

When you start a new project, you must create a new GitHub repository. The easiest way is to:

1. Log in to your GitHub account.
2. Click on **New**



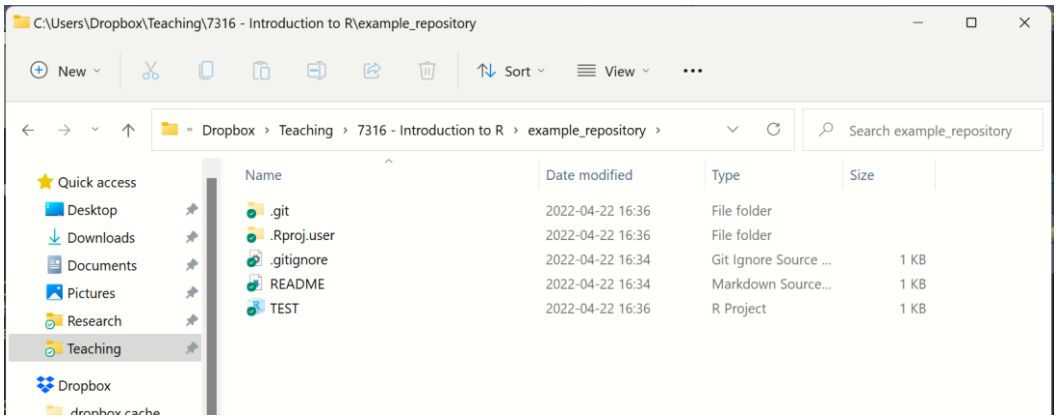
Create a new repository.

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 allow can see your repository).
5. In *Add .gitignore*, choose `R` (if you are creating an R project).
6. You can also 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 populate it with files.

6.3.3 Managing `.gitignore`

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



Content of your local repository

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

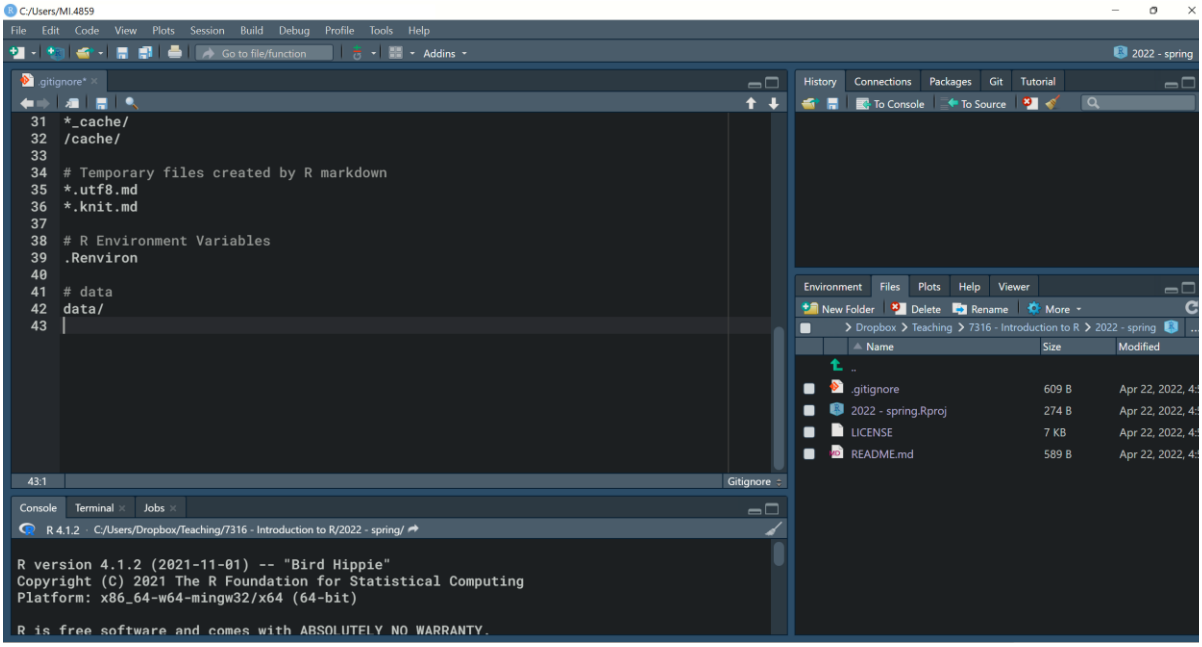
- Your repository contains a file named `.gitignore`. This is a text file listing all the files you want to ignore from the version control (this means that the files and folders listed here will **NOT** be committed in the version control nor pushed to Github).
- To ensure your data do not end up somewhere 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

---

[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.

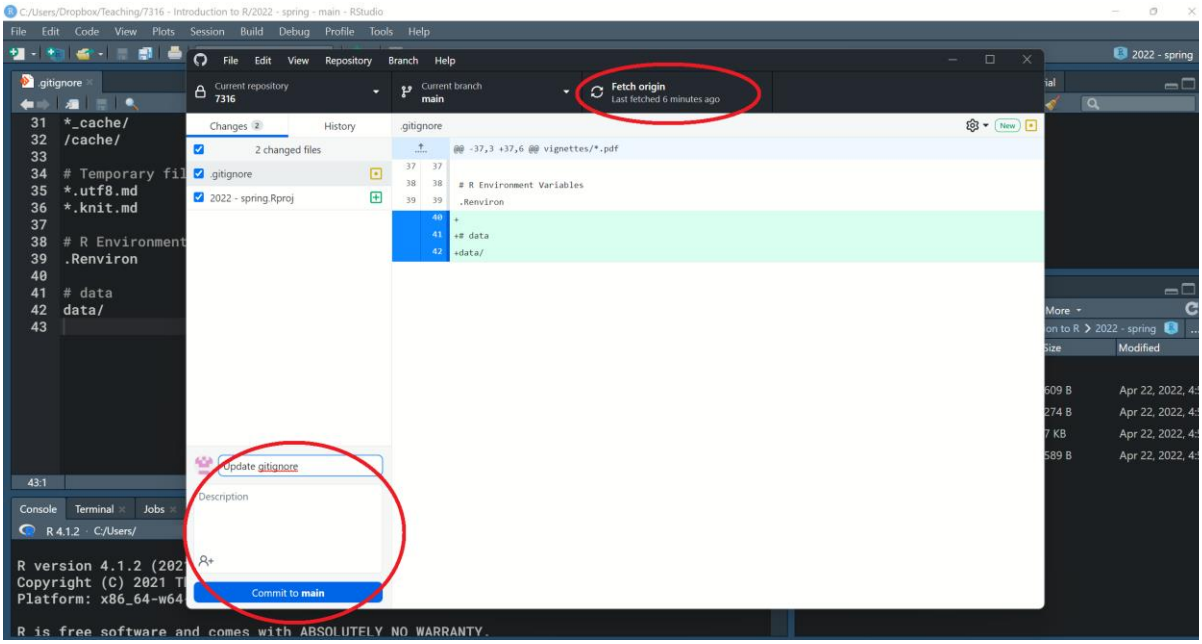
- You can also do that with any other file or folder in your local repository where you do not want to track changes.



The .gitignore file.

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 changed.



Commit a change

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

6.3.5 Syncing changes with a remote repository

Commits are sufficient to manage versions if you track changes with a local repository. Still, if you are using version control with a remote (*i.e.,* online) repository, you will need to sync your commit and the commits of others with GitHub. For this, you have one button I highlighted in red in the screenshot above: **fetch origin**. This button has three states:

1. **fetch origin** checks if any changes have 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 team members are asynchronous (*i.e.,* you do not need to be connected to the internet to make local changes, but you will have to push them to the cloud later on). You may sometimes manage version conflicts if multiple users simultaneously modify the same chunk of code (i.e., lines of code). Git helps you with that.

6.3.6 Viewing previous commits

- Click on the History button to view previous versions of the files (along with annotations supplied with the commit message).
- From there, you can see the differences between versions and 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 copy over the file with code from the previous commit.

7. Code style and elegance

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

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 style guides suggest an 80-character 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.

7.2 Give air to your code!

It may seem useless when you start, but having adequately designed code increases readability and reduces 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 have thousands of lines of code (it comes faster than you think...), one style will give you headaches; the other won't.

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 the R interpreter will NOT interpret. 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`.

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 can 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, commas, 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.