WJP - R Coding Handbook

Carlos A. Toruño P.

11/1/22

Table of contents

Introduction

Figure 0.1: Zacatal by nicaraguan painter Raúl Marín

This coding handbook is a continuous work maintained by the Data Analytics Unit (DAU) from The World Justice Project (WJP) with the aim of unifying the different aspects of the work carried by the DAU. In this book, you will find not only general guidelines but also several issues that will help the reader to understand and contribute to our daily tasks.

The handbook cover aspects related to the general workflow, the coding process and the visualization guidelines used by the team. As mentioned earlier, this book is a continuous work in progress and, as such, the rules and guides are not written in stone but rather, subject to improvements that every member of the team is open to discuss and include in this material.

Prerequisites

In order to assimilate the content in this chapter (and the entire handbook) we will require you to have the following:

- An intermediate knowledge of R Programming Language.
- R Studio is installed in your computer. However, you are free to use any other Integrated Development Environment (IDE) or Text Editor of your choice.
- A basic knowledge of Git.
- A GitHub account.
- Access to the DAU's SharePoint.
- Although is not required, we strongly advise you to install GitHub Desktop.

If you are new to R, I would recommend you to check the Hands-On Programming with R book written by Garrett Grolemund. If you are already comfortable with the R programming language but you need an introduction on how to analyze data using it, I would suggest you to read the R for Data Science book edited by the R4DS team.

Handbook Structure

The content in this Handbook is organized as follows:

- **Part I: Workflow**: A brief chapter in which we will cover the basic guidelines and issues related to the DAU workflow when programming with R and R Studio along with the basic setup.
- **Part II: Coding**: The main chapter of the handbook where we discuss the main aspects of the coding style when working in projects related to the DAU.
- **Part III: Viz Aesthetics**: Manual of style, style guidelines, pre-defined settings and other aspects related exclusively to data visualization are presented in this chapter.
- **Appendix**: Final part of the handbook where you can read about R-related content for the daily tasks that you will have to face as a member of the DAU.

Licence

This website is free to use and it is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License. Physical copies of this handbook are not currently available. If you are not a member of the DAU team, feel free to report typos, request information or even contribute by commenting or leaving a pull request at ctoruno/DAU-R-Style-Manual. This handbook was created using Quarto Publishing System.

To learn more about the work of The World Justice Project, please visit the official website.

1 Workflow

In this chapter, we will cover the basic guidelines and issues related to the prefered workflow when programming with R programming language. This chapter will cover four different aspects: a) *Cloud Storage*, b) *Version Control System*, c) *R Studio Projects*, and d) *File Management practices.*

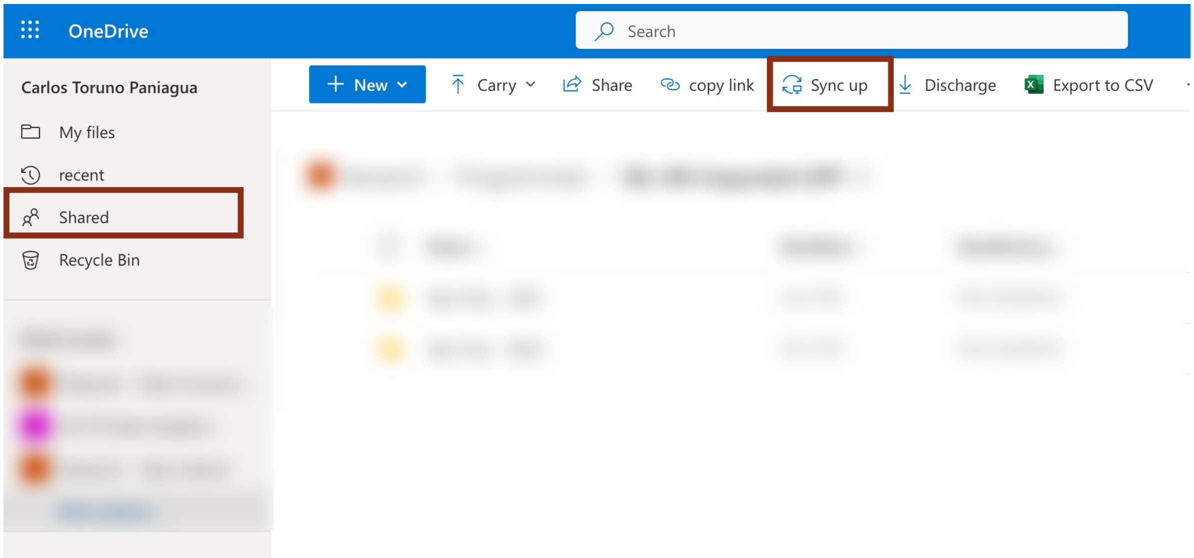
1.1 SharePoint

All files used by the DAU are stored in the Data Analytics folder within the organization's SharePoint. This is done in order to provide an easy access to all members of the team. As a rule, you are required to work and modify these files directly from this online directory. To achieve this, you will need to download the OneDrive app in your computer and sign in using your WJP account. If you are already using One Drive with your personal account, you can add an additional professional account which will function parallel to your personal account in your computer. For more information, see the following website.

Given that the DAU SharePoint is a directory created and administered by the organization, you will have to sync this directory to your own OneDrive WJP Account. In order to achieve this, follow these steps:

1. Login into your WJP Outlook account in any web browser.
2. Once into your account, access ***OneDrive***. This will open a new tab for you.
3. In the left side panel, locate the ***Share with Me***tab and click it.
4. Locate the ***Data Analytics folder***.
5. Once that you are inside the folder, you will see the ***Add Shortcut to My Files***,or, ***Sync Up*** option in the upper toolbar, depending on your OneDrive version.
6. Follow the instructions and sync it to your WJP folder in your computer.

By syncing your work with the SharePoint, the whole team is going to be able to see and review changes as they go. However, if more than one person is working on the same file at the same time, it is likely that this working flow will produce several mistakes. Therefore, additionally to this shared cloud storage system, we need a version control tool that will allow us to modify and collaborate in team projects without these kind of issues. Given that most of our work is focused in coding, we use ***Git Version Control*** and the ***GitHub*** platform for this.



1.2 Git

Git is a free and open source software that allows users to set up a version control system designed to handle projects. Given the nature of its features, it is normally used for collaboratively developing code and data integrity. GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code. For a gentle introduction to Git and GitHub, see the following post published by Kinsta. For a more in depth introduction, please refer to the GitHub documentation or watch this video tutorial.

Using the Git features allow us to simultaneously work on the same project and even in the same code without worrying about interfering with other members of the team. As a rule, every project carried by the DAU has a code administrator who is in charge of setting up the GitHub repository and add other members of the team as project collaborators. Additionally, the code administrator is in charge of setting the *main branch* and the initial structure of the code (see the [data-management](https://ctoruno.quarto.pub/wjp-r-handbook/workflow.html#data-management) section on this chapter). It is required that the GitHub repository have its *main branch* in its respective SharePoint folder.

> **Note:** It is highly recommended to create the repository from GitHub.com and not from the local machine to avoid the initial commit that can include system files such as **.DS_STORE** files

We use convergence development[1] to collaboratively code in the same project. For this, it is

[1] For more information about convergence development and branches, we advise you to refer to this article from Pluralsight.

highly recommended that each team member works on a separate *branch* and, once the data routine is done, the auxiliary branches can be merged into the main branch of the repository. Collaborators that are not the code administrator have the option to clone the GitHub repository in their computer in a local directory that it is not sync to the SharePoint and work in their respective branch from a local copy outside the SharePoint. In other words, it is only the functional final version contained inside the *main branch* the one that it is going to be sync in SharePoint.

> **Important:** GitHub is used to keep track of the code we use in each project. Under no circumstance, we will include the data sources in the online repositories.

1.3 Projects

When you load a data set or source an R script, you will have to set up the working directory where these files are located in your computer. However, the path to this working directory is quite different for all the members of the team. R Studio allow us to enclose all of our analysis, code and auxiliary files into a project.

A project is a feature that allow us to work with the analysis we are carrying without having to worry about where does these files are stored or who is working on them. Say goodbye to `setwd("...")`. Besides managing relatives paths, R projects allow users to keep a history of actions performed and even keep the objects in your environment. Because of this, projects are the cornerstone of our work when performing analysis with R.

As a rule, every project has a file named `project-name.Rproj` in its root directory and open it should be the first action when working on a project. For more information on working with R projects, refer to the Workflow section from the R for Data Science book.

1.4 SharePoint Global Access

As mentioned before, we try to keep everything within the Project's wworking directory. Nevertheless, due to their nature, sometimes we will have to load file from outside this directory. For example, we will constantly loading the GPP or QRQ datasets or the WJP font files. Instead of copying these files into the projects directory, we access them through what we call a Global Access. These files are located in a static path inside the DAU SharePoint. Assuming that you have installed the desktop version of OneDrive, you can access these files quite easily. However, this path is unique in every local machine. Therefore, it is very common to find a variable named `path2SP` the following lines of code in the settings of every project:

```
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
##
## 2.  SharePoint Path                                                      ----
##
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# SharePoint path
if (Sys.info()["user"] == "johnDoe") {
  path2SP <- "/Users/johnDoe/OneDrive-WJP/Data Analytics/"

} else if (Sys.info()["user"] == "anaPerez") {
  path2SP <- "/CloudStorage/OneDrive-WJP/Data Analytics/"

} else if (Sys.info()["user"] == "kumikoNagato"){
  path2SP <- "/Users/kumikoNagato/Library/OneDrive-WJP/Data Analytics/"

} else{
  path2SP <- "PLEASE INSERT YOUR PERSONAL PATH TO THE SP DAU DIRECTORY"

}
```

This way, loading or accessing files that are outside the Project's directory but inside the DAU SharePoint is possible without adjusting the code everytime a different user tries to run the code. For example, if you would like the user to load the WJP fonts, you simply have to paste their location within the SharePoint and the `path2SP` string to let the code run smoothly between users.

```
path2fonts<- paste0(path2SP, "6. Country Reports/0. Fonts/")
font_add(family     = "Lato Full",
         regular    = paste0(path2fonts, "Lato-Regular.ttf"),
         italic     = paste0(path2fonts, "Lato-LightItalic.ttf"),
         bold       = paste0(path2fonts, "Lato-Bold.ttf"),
         bolditalic = paste0(path2fonts, "Lato-BoldItalic.ttf"))
```

1.5 Data Management

The University of California San Diego (UC San Diego) has a Data Management Best Practices that reviews common guidelines for managing research data. In this handbook, I will focus on two topics mentioned by those guidelines: File Organization and Documentation.

1.5.1 File Organization

The file organization involves two important elements: filing system and naming conventions.

A filing system is basically the organization structure (directories, folders, sub-folders, etc) in which files are stored. There are no standard rules about how this should be done. However, the chosen filing system needs to make sense not only to the person currently working on a given project but to anyone going through these files in the future. As a rule, each project would have the following sub-folders:

- Code: Depending on the complexity of the project, you could choose to create separate directories within the Code folder for Stata, R or Python files.
- Data: Depending on the complexity and nature of the project, you could choose to create separate directories within the Data folder for RAW, INTERMEDIATE or CLEAN data sets.
- Outcomes: The outcomes of the project might have several different formats. For example, images could be in PNG and/or SVG format, Reports might be created in PDF, some tables might be exported as TEX files, etc. The outcomes folder should have a separate directory for each one of these formats.

In some cases, the creation of a PDF report is key, for example, the Regional Country Reports. For these projects, we strongly advise to create a separate directory to store the code files used for the report. At the moment of writing this handbook, these reports are created using R Markdown, but a migration to Quarto is feasible in the future.

- Markdown/Quarto

In relation to the naming conventions, these are a set of rules designed to complement the filing system and help collaborators in understanding the data organization. Each project have the flexibility to use a specific set of naming rules to use in the filing system. However, there are a few general rules to note:

- Use descriptive file names that are meaningful to you and your colleagues while also keeping them short.
- Avoid using spaces and make use of hyphens, snake_case, and/or camelCase.
- Avoid special characters such as $, #, ^, & in the file names.
- Be consistent not only along the project but also across different projects. If all different data files and routines are named in the same way, it's easier for you to use those tools across projects and re-factor routines.

1.5.2 Documentation

When initializing the GitHub repository, we strongly suggest to include a README file with it. Such file should be a Markdown document including the following elements:

- A brief description of the project and the role of the DAU in it.
- Referred person contact, which should the project leader and the code administrator.
- A brief description of the filing system along with what to find in each sub-directory.
- Given that no data is uploaded into the online repositories, include the following descriptions:
  - Data needed to run the code: source and process to obtain it.
  - Location of the data in the SharePoint.
- A brief description on how to read the code with, if possible, visual aids.

For an example on one of these files, please check the following README file. If possible, use this example as a template for future projects.

Depending on the complexity of each project, the team can add any suitable documentation files if needed. Additionally, the DAU keeps records of significant issues encountered during each project in order to facilitate the solution process in future projects. These records are kept in the Issues Logbook of the DAU.

2 Coding

In this chapter, we will cover some basic guidelines and styling rules related to how the coding is done at the DAU-WJP. When programming data analysis routines, it is very hard to vanquish the personal style that every person has. Therefore, this chapter is focused on giving general guidelines that will allow to standardize their codes for an easy collaboration among all team members.

Remember, when writing a code, you are just the author not the audience. Therefore, think on how would other people understand what you are writing without the ideas and knowledge you have at the moment of creating your code. Writing comments, using titles as step-by-step guides, documenting the issues, all of these actions will greatly help other team members to understand what you have done and why you have done it. Also, this will help you in the future to understand what you did in the past and reduce the level of dependency of a given project on its collaborators.

2.1 Script Headline and Outline

The headline is very important because it gives the general information about the script, its purposes, the authors, the program version, among other important details. Within the DAU, we have the following template:

```
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
##
## Script:              PROJECT NAME - Script Purpose
##
## Author:              Author 1 Name   (email)
##                      Author 2 Name   (email)
##
## Creation date:       Month Day, Year
##
## This version:        Month Day, Year
##
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

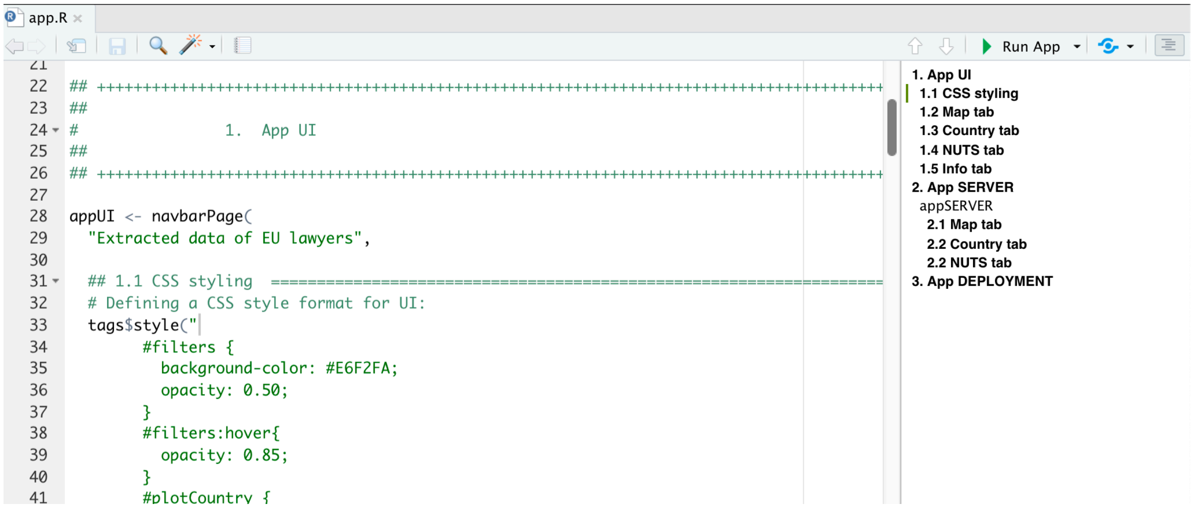
As you can see, the template highlights the most important information that we need to know when opening a script. This information should give any team member a general idea on the project status even if this person has never collaborated in the project before. However, this is just a first step in the process. Our template goes beyond and it also displays how to use Titles, Subtitles and Steps within the script:

```
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
##
#                1.  TITLE
##
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


## 1.1 SUBTITLE  =========================================================================

# Step 1

# Step 2

## 1.2 SUBTITLE  =========================================================================

# Step 1

# Step 2
```

Unlike the Headline seen above, the use of this outline to structure your code will vary from script to script. For example, if you are working on a very short routine of less than 100 lines. You might not need to use titles and subtitles and you would rather choose to only use steps. Therefore, the extent to which this outline is feasible will be a decision of the code administrator.

The main objective of this outline is to provide an organized outline for the script. The utility of this outline increase with the complexity of the code. Although we strongly advice you to avoid very long routines (we will come back to this when we talk about refactoring and modules), we understand that sometimes the code might be very complex by nature. As an example, look at the following script from the Shiny App that we prepared for the EU Project. Even though the code extends for less than 500 lines, it is difficult to read due to the ***reactivity*** and ***laziness*** concepts that come along when programming a Shiny App.

If you want to knowww more about R scripts and their outlines in RStudio, you should definitely check out this post by Nate Day about How to own outlines in RStudio.



```
## ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
##
#                1.  App UI
##
## ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

appUI <- navbarPage(
  "Extracted data of EU lawyers",

  ## 1.1 CSS styling  ==================================================================
  # Defining a CSS style format for UI:
  tags$style("
        #filters {
          background-color: #E6F2FA;
          opacity: 0.50;
        }
        #filters:hover{
          opacity: 0.85;
        }
        #plotCountry {
```

2.2 Loading Packages

On any routine, the first thing is to load the packages that you will be using in the code. Usually, you will see routines calling the `library` or `request` modules to load the needed packages. However, given that this is a team collaboration, not everyone have these packages installed or, even if they have it, the script requires a certain version of the package in order to run. Therefore, we need to, not only load, but also check for these other requirements as well.

W=To achieve this, we could write down a series of if conditions. However, we choose to rely on the pacman package to do this. As such, we only request that every team member has this package installed in their local machine. If every team member has it installed, it is quite easy to load the required libraries by calling the `p_load` function. Take a look at the following example:

```
# Required packages
library(pacman)

# Development version
p_load_gh(char = c(

  # Visualizations
  "xl0418/ggradar2", "davidsjoberg/ggsankey",

  dependencies = T
))
```

```r
# Stable CRAN release
p_load(char = c(

  # Visualizations
  "showtext", "ggtext", "ggsankey", "ggrepel", "ggplotify",
  "gridExtra", "ggradar2", "patchwork",

  # Data Loading
  "haven", "readxl",

  # Other
  "margins",

  # Good 'ol Tidyverse
  "tidyverse"

))
```

Three things are worth noticing. First, if some packages require the development version, we use the `p_load_gh` to install their latest release from GitHub. Second, we always install the development releases first and then the stable CRAN version at last. This is done in order to leave the tidyverse for last and avoid any other package to mask over the tidyverse functions. Third, for very complex routines, we might need to load several packages, it is strongly recommended to comment on the utility and need for each package so all team members can have a general idea on their use before reading the respective documentation.

2.3 Coding Style

2.3.1 The Tidyverse Style guide

In the DAU, we strongly rely on the guidelines defined by *Hadley Wickham* in its Tidyverse Style Guide. We request that every DAU member read this style guide before collaborating with other team members in a given project. As stated in the first page of the guide:

> *Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread.*

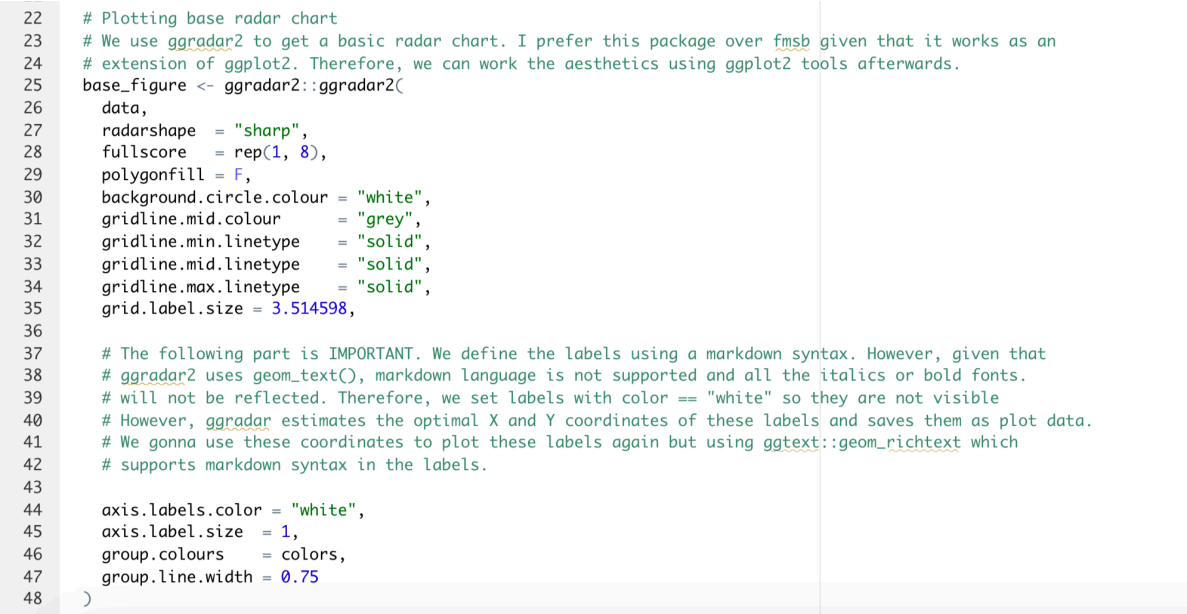
This guide is, as its name suggest, just a guide. As such, some of its guidelines might not be consistent with the coding style that the DAU as a whole have. Therefore, we do not rely on the use of styler and lintr packages to style our codes. In what follows, we will highlight some topics and also complement with some of our own style guidelines the aforementioned reference.

2.3.2 Commenting

Commenting is one of the most important elements of coding. However, the excessive use of comments might also reflect an issue. The outline presented at the beginning of this chapter introduced the use of titles, subtitles and steps. Under this structure, ***steps*** *are brief comment lines used to guide the reader on the utility and objective of different chunks of code.* Take a look at the following example and how steps are guiding the reader along the code.

```r
# Plotting each panel of Figure 4
imap(c("A" = "q18",
       "B" = "q21",
       "C" = "q33",
       "D" = "q73",
       "E" = "q74"),
     function(var4plot, panelName) {

       # Filtering data2plot to leave the variable for each panel
       data2plot <- data2plot %>%
         filter(group %in% var4plot)

       # Applying plotting function
       chart <- LAC_divBars(data           = data2plot,
                            target_var     = "perc",
                            grouping_var   = "country",
                            diverging_var  = "status",
                            negative_value = "Negative",
                            colors         = colors4plot,
                            labels_var     = "label")

       # Saving panels
       saveIT.fn(chart  = chart,
                 n      = nchart,
                 suffix = panelName,
                 w      = 100.8689,
                 h      = h)
     })
```

In a very ideal scenario, the code just need to efficiently use ***steps*** in order to be clear. However, we understand that we might face complex situations that need the use of extra lines of comments in order to be clear. Under this scenario, please use comments to explain the ***why*** and not the *what* or *how*. For example, look at the following example from the WJP Data Viz repository:



```
# Plotting base radar chart
# We use ggradar2 to get a basic radar chart. I prefer this package over fmsb given that it works as an
# extension of ggplot2. Therefore, we can work the aesthetics using ggplot2 tools afterwards.
base_figure <- ggradar2::ggradar2(
  data,
  radarshape  = "sharp",
  fullscore   = rep(1, 8),
  polygonfill = F,
  background.circle.colour = "white",
  gridline.mid.colour      = "grey",
  gridline.min.linetype    = "solid",
  gridline.mid.linetype    = "solid",
  gridline.max.linetype    = "solid",
  grid.label.size = 3.514598,

  # The following part is IMPORTANT. We define the labels using a markdown syntax. However, given that
  # ggradar2 uses geom_text(), markdown language is not supported and all the italics or bold fonts.
  # will not be reflected. Therefore, we set labels with color == "white" so they are not visible
  # However, ggradar estimates the optimal X and Y coordinates of these labels and saves them as plot data.
  # We gonna use these coordinates to plot these labels again but using ggtext::geom_richtext which
  # supports markdown syntax in the labels.

  axis.labels.color = "white",
  axis.label.size   = 1,
  group.colours     = colors,
  group.line.width = 0.75
)
```

In this example, we need to explain ***why*** did we choose to use ggradar2 to plot a base figure and then complement the resulting plot using ggplot2, instead of just relying on ggplot2. Given that this is a uncommon course of action, we give a detailed explanation of our reasons. This is a good example on when and how to use comments to complements ***steps*** in a script. Nevertheless, the excessive need of using this long commenting lines might suggest that e need to modify or re-write our code to make it clearer.

Finally, it is worth to mention that you do not meed to be afraid of breaking long pipe sequences in order to add a ***step*** in between pipes in order to make your code easier to read. Take a look at the following example in which we introduced ***steps*** within the `mutate()` pipe in order to make our code easier to read and understand.

```
# Defining data frame for plot
  data2plot <- data_subset.df %>%
    filter(country == mainCountry & year %in% yrs) %>%
    select(year, starts_with("q49"), EXP_q23d_G1) %>%
    mutate(

      # We need to concatenate variables q49d_G1 and EXP_q23d_G1 into a single one
      q49d_G1_merge = rowSums(across(c(q49d_G1, EXP_q23d_G1)),
                              na.rm = T),
      q49d_G1_merge = if_else(is.na(q49d_G1) & is.na(EXP_q23d_G1),
                              NA_real_,
                              q49d_G1_merge),

      # Transforming everything into binary variables
      across(!year,
             ~if_else(.x == 1 | .x == 2, 1,
                      if_else(!is.na(.x) & .x != 99, 0, NA_real_)))
    ) %>%
    select(!c(q49d_G1, EXP_q23d_G1)) %>%
    group_by(year) %>%
    summarise(across(everything(),
                     mean,
                     na.rm = T)) %>%
    rename(group = year)
```

2.3.3 Line lenght and breaks

In the script headline and outline that we presented above, as you could have observed, the titles and subtitles were accompanied by dividing lines such as:

```
## +++++++++++++++++++++++++++++++++++++++++++++++++++++++++

## =========================================================
```

As explained in this post, the main objective of using these lines is to organize our script outline in RStudio. However, they have another function in our workflow and it is to set the line lenght for the entire script.

As we know, it is easier to read code vertically than horizontally. That's why most coding guides stablish a line length for aesthetics and if you are writing a line of code that exceeds this lenght, it is ***highly recommended*** to break the line. Within the DAU, we have established a line length of 110 characters, which is exactly the length of the dividing lines in our template. As a rule, ***not a single line of code*** should be longer than the dividing lines. If by any reason, your line of code exceeds it, you should break into multiple lines. There are multiple ways to achieve this, here we just provide a few examples and guidelines about it.

2.3.3.1 Use one argument per line

Try to always break the line after every comma so you can leave one argument per line. However, more than a rule, this should be considered a guide and in some case it might be better to omit this guide for aesthetics purposes. For example, take a look at this example:

```
# Defining data frame containing the info to plot
  data2plot <- data_subset.df %>%
    filter(country == mainCountry & year == latestYear) %>%
    select(CAR_q59_G1,
           CAR_q59_G2,
           unlist(vars4plot,
                  use.names = F)) %>%
    mutate(
      govSupp = case_when(
        CAR_q59_G1 == 1   | CAR_q59_G2 == 1   ~ "Gov. Supporter",
        CAR_q59_G1 == 2   | CAR_q59_G2 == 2   ~ "Non Gov. Supporter",
        CAR_q59_G1 == 99  | CAR_q59_G2 == 99  ~ NA_character_,
        is.na(CAR_q59_G1) & is.na(CAR_q59_G2) ~ NA_character_
      ),
      across(!c(CAR_q59_G1, CAR_q59_G2, govSupp),
             ~if_else(.x == 1 | .x == 2, 1,
                      if_else(!is.na(.x) & .x != 99, 0,
                              NA_real_)))
    ) %>%
    group_by(govSupp) %>%
    filter(!is.na(govSupp)) %>%
    summarise(across(everything(),
                     mean,
                     na.rm = T)) %>%
    pivot_longer(!govSupp,
                 names_to   = "category",
                 values_to  = "value2plot")
```

Take a look at how I decided to keep one argument per line in the `select()`, `summarise()` and `pivot_longer()` functions, but not for the `if_else()` function within the `mutate()` call or when supplying multiple arguments in the form of a vector `c()` within the `across()` call. Sometimes, is easier to read or even aesthetically better if you keep short arguments and calls in a single line.

2.3.3.2 Break long calls

I think that at this point, it is more than obvious that if you are creating new variables using `mutate()` or `summarise()`, you should leave each call in a new line. However, what it is not that obvious is that breaking the line right after you call the function can improve the readability of your code by a lot. Take a look at the `mutate()` in the previous example. We break the line after the opening parenthesis in that *mutate*, we generate the new variables, and then we leave the closing parenthesis alone in a new line at the end. This is a trick that I learned from HTML and that it helps a lot when you have very long sequences within a function. This tip also applies to other functions that usually require long sequences of arguments such as `case_when()`, which you can also check in the example above.

2.3.3.3 paste() is your best friend when dealing with long strings

There are a few times in data analysis when you face long strings and you have to deal with it. Take a look at the following example in which we had to write down long labels in HTML. We decided to leave the ***tags*** (, , etc) in separate lines so the reader could easily distinguish between CSS attributes and visible text.

```
# Defining labels - Part II: Percentages + text as HTML
  applying_labels.fn <- function(text = text, color_code, value_vectors){
    case_when(
      text == 'q7' ~ paste("<span style='color:black;font-size:6.3mm;font-weight:bold'>",
                           value_vectors["q49a"],
                           "</span>",
                           "<br>",
                           "<span style='color:#524F4C;font-size:3.5mm;font-weight:bold'>"
                           "Is effective in bringing<br>people who commited<br>crimes",
                           "</span>"),
      text == 'q8' ~ paste("<span style='color:black;font-size:6.3mm;font-weight:bold'>",
                            value_vectors["q49b_G2"],
                            "</span>",
                            "<br>",
                            "<span style='color:#524F4C;font-size:3.5mm'>",
                            "Ensures equal treatment<br>of victims by allowing all<br>",
                            "victims to seek justice<br>regardless of who they are",
                            "</span>")
    )}
```

2.3.4 Indentations and column alignment

Data analysis usually comes packed with large chunks of code that are easy to read for the author. However, for someone that it is not the author of these lines, it might be difficult to read these large chunks of codes. Therefore, as authors, we need to make it easy for the audience to read our code. We already talked about adding comments, line breaks and other similar tasks that can increase the readability of our code. One more thing to take into account is the visual aesthetics of our code. *Clean and tidy code is easier to read and follow for someone* *who is not familiar with it.* Small changes to your code can greatly improve its readability. One of such small changes is to properly use **indentations** to **align collumns**.

Let's try something. Take a look at the following `ggplot theme` definition and try to locate the font face defined for the text in the Y-Axis:

```
WJP_theme <- function() {
  theme(panel.background = element_rect(fill = "white", size = 2),
        panel.grid.major = element_line(size = 0.25, colour = "#5e5c5a", linetype = "dashe
        panel.grid.minor = element_blank(),
        axis.title.y = element_text(family = "Lato Full", face = "plain", size = 3.514598*
                                    margin   = margin(0, 10, 0, 0)),
        axis.title.x = element_text(family = "Lato Full", face="plain", size = 3.514598*.p
                                    color = "#524F4C", margin=margin(10, 0, 0, 0)),
        axis.text.y = element_text(family = "Lato Full", face = "plain",
                                   size = 3.514598*.pt, color = "#524F4C"),
        axis.text.x = element_text(family = "Lato Full", face="plain", size=3.514598*.pt,
        axis.ticks= element_blank(),
        plot.margin = unit(c(0, 0, 0, 0), "points")
  )
}
```

Now, take a look at the same theme definition and try to locate the same argument I told you before, but now, with a tidily organized code:

```
WJP_theme <- function() {
  theme(panel.background   = element_rect(fill = "white",
                                          size = 2),
        panel.grid.major   = element_line(size     = 0.25,
                                          colour   = "#5e5c5a",
                                          linetype = "dashed"),
        panel.grid.minor   = element_blank(),
        axis.title.y       = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C",
                                          margin   = margin(0, 10, 0, 0)),
        axis.title.x       = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C",
                                          margin   = margin(10, 0, 0, 0)),
        axis.text.y        = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C"),
        axis.text.x        = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C"),
        axis.ticks   = element_blank(),
        plot.margin  = unit(c(0, 0, 0, 0), "points")
  )
}
```

Which one was easier for you?

In the previous example, the difference might appear to be small. However, the advantage of using a tidy and clean way to write your code will exponentially increase with the length and complexity of your code. Locating an element in a chunk of 30 lines is easier than locating an element in a chunk of 300 lines.

2.4 Re-factoring and modular programming

Writing code in a proper way is not only about how easy to read is your code, but also, how easy it is to maintain. If you need to change something, you should be able to do it cleanly and quickly. Nevertheless, most of the time, these two goals are not aligned and it is your job to achieve an optimal balance.

Take a look at the following code in which we are defining a text to be in HTML format:

```
data_modified <- data %>%
  mutate(
      label =  case_when(
        text == 'q49a' ~ paste("<span style='color:", color_code, ";font-size:6.326276mm;f
                               value_vectors["q49a"],
                               "</span>",
                               "<br>",
                               "<span style='color:#524F4C;font-size:3.514598mm;font-weigh
                               "Is **effective** in bringing<br>people who commit<br>crime
                               "</span>"),
        text == 'q49b_G2' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49b_G2"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Ensures **equal treatment<br>of victims** by allowing a
                                  "victims to seek justice<br>regardless of who they are",
                                  "</span>"),
        text == 'q49e_G2' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49e_G2"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Safeguards the<br>**presumption of<br>innocence** by tr
                                  "accused of<br>crimes as innocent<br>until proven guilty
                                  "</span>"),
        text == 'q49c_G2' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49c_G2"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Ensures **equal treatment of<br>the accused** by giving
                                  "fair trial regardless of who<br>they are",
                                  "</span>"),
        text == 'q49e_G1' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49e_G1"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Gives **appropriate<br>punishments** that fit<br>the cr
                                  "</span>"),
        text == 'q49d_G1_merge' ~ paste("<span style='color:", color_code, ";font-size:6.3
                                        value_vectors["q49d_G1_merge"],
                                        "</span>",
                                        "<br>",
                                        "<span style='color:#524F4C;font-size:3.514598mm'>
                                        "Ensures **uniform quality** by<br>providing equal
                                        "regardless of where<br>they live",
                                        "</span>"),
        text == 'q49c_G1' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49c_G1"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Ensures everyone<br>has **access** to the<br>justice sy
                                  "</span>"),
        text == 'q49b_G1' ~ paste("<span style='color:", color_code, ";font-size:6.326276m
                                  value_vectors["q49b_G1"],
                                  "</span>",
                                  "<br>",
                                  "<span style='color:#524F4C;font-size:3.514598mm'>",
                                  "Ensures **timeliness**<br>by dealing with<br>cases prom
                                  "and<br>efficiently",
                                  "</span>")
      )
  )
```

What will happen if, out of nowhere, the design team ask you to increase the font size from 3.514mm to 3.758mm? They are asking for a single change, but how many time would you have to modify that font size? What if you make a mistake or you skip one of those changes? As you can see, this is an easy to read code, but it ends up being very hard to maintain.

Let's review this code again and see how can it be improved. The first thing that we can notice is that we are repeating the same 5 lines several times. As I once read on the internet:

> **When you've written the same code 3 times, write a function.**

Let's do exactly that:

```
data_modified <- data %>%
  mutate(

    # Converting labels into HTML syntax
    across(label,
           function(raw_label){
             html <- paste0("<span style='color:", color_code,
                            ";font-size:6.326276mm;font-weight:bold'>",
                            value_vectors[text],
                            "</span>",
                            "<br>",
                            "<span style='color:#524F4C;font-size:3.514598mm'>",
                            raw_label,
                            "</span>")
             return(html)
           })
  )
```

This way, the code has become shorter and easier to maintain. If we need to change the font size, we do it just one time. Both chunks of code have the same result, but we change the way they produce that result. This process is called **re-factorization**. To learn more about this and all the re-factoring methods, you can check refactoring.guru.

Another complementary way to keep your code tidy and easy to maintain is to reduce the length of your script by *splitting it into multiple scripts according to their functionality* or **modules**. This process is called **modular programming**.

A script of more than a thousand lines might be very difficult to maintain in Data Science. Therefore, it is very advisable to split it into several modules. For example, if you are working on a very long script, you can group all the lines where you load all the required settings and data and save it into a module called `settings.R`, and you can `source()` it from your main script. Look at the following example where we source all the different modules from a local directory called `R` and even directly from GitHub:

```
# Required Packages, Fonts, ggplot theme, color palettes, comparison countries and other g
# loaded from the following script:
source("Code/settings.R")

# Loading functions for sections
source("Code/S01.R")
source("Code/S02.R")
source("Code/S03.R")

# Loading plotting functions from GitHub
source("https://raw.githubusercontent.com/ctoruno/WJP-Data-Viz/main/loading.R")
```

Breaking your code into modules makes your code easier to read because it means separating it into smaller pieces that only deal with one aspect of the overall functionality. By doing this, your are making your code:

- Easier to test
- Easier to find things later
- Easier to re-use
- Easier to re-factor
- Easier to collaborate with

Overall, it has a lot of advantages in comparison to monolithic and overwhelming code. Nonetheless, if you overuse it without a logic behind, it can end up making your code messy. As a result, you need to be careful when making modular programming.

2.5 Issues Logbook

As mentioned before, the DAU team keeps records of significant issues encountered during each project in order to facilitate the solution process in future projects. These records are kept in the Issues Logbook of the DAU.

As a rule, the Issues Logbook is a GitHub repository maintained by the DAU team. It is organized by relevant topic or by project name. As a rule, each issue record should be saved as a *Markdown file* using a short name that describes its content and substituing spaces for hyphens. For example: `Pandoc-variable-in-YAML.md`.

The markdown file should follow the next structure: 1. Team member that identified the issue. 2. Short description of the problem or issue. 3. Short description of the solution (if any). 4. Additional notes (if any).

You can use this template for recording issues. For an example of an actual issue recorded, you can check this example.

3 Visualization Aesthetics

In this chapter, we will briefly explain the different resources you have available and that you should make use of when working of visualizations as a member of the DAU.

3.1 WJP Style Manuals

2021

Brand Standards Manual

World Justice Project

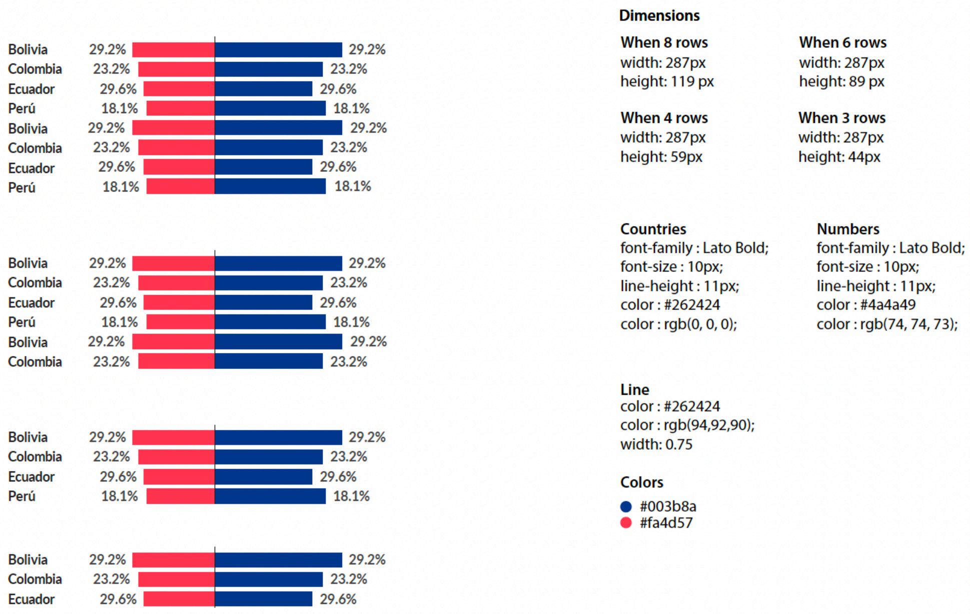
The WJP Brand Standard Manual, covers two major areas. First, the **Design Style Guide** which cover issues such as Color Palette, Fonts and Layout. However, these issues are more related to the work that is generally performed by the communications and design team. The second area is the **Data Viz Style Guide** and it covers issues such as the color palettes, fonts and layouts that should be used in WJP's charts and tables.

The **Data Viz Style Guide** is complemented by the Color Palette Manual. This document extends on the use and context of the different color palettes available for WJP.

The application of these issues is mandatory when working with data visualizations related to the Rule of Law Index but they should be considered only as guidelines when working on other projects such as the Country Reports or the EU Sub-national.

The specifications of the plots to be published are defined by the communications and design team and are to be implemented by the DAU. The design team will assemble a set of specifications that you should follow and incorporate in your charts and tables. See the following example from the 2022 Country Reports in Latin America and the Caribbean:

FIGURE 4



Dimensions

When 8 rows
width: 287px
height: 119 px

When 6 rows
width: 287px
height: 89 px

When 4 rows
width: 287px
height: 59px

When 3 rows
width: 287px
height: 44px

Countries
font-family : Lato Bold;
font-size : 10px;
line-height : 11px;
color : #262424
color : rgb(0, 0, 0);

Numbers
font-family : Lato Bold;
font-size : 10px;
line-height : 11px;
color : #4a4a49
color : rgb(74, 74, 73);

Line
color : #262424
color : rgb(94,92,90);
width: 0.75

Colors
- ● #003b8a
- ● #fa4d57

3.2 Predefined ggplot themes

In order to achieve the desired plot that fits the specifications sent by the design team, we will have to manually set the colors, the font family, the font sizes, among other many items. To avoid doing this repeatedly for every chart in a project, it is common to set up a predefined ggplot theme for a given project. This predefined theme will not suit every single chart of the project but it should -ideally- minimize the use of the `theme()` function to set up the desired specifications.

As a rule, the ggplot theme to be used in each project is defined in the `settings.R` file. To do this, you first define the ggplot theme as follow:

```
# Defining a ggplot WJP theme
WJP_theme <- function() {
  theme(panel.background   = element_blank(),
        plot.background    = element_blank(),
        panel.grid.major   = element_line(size     = 0.25,
                                          colour   = "#5e5c5a",
                                          linetype = "dashed"),
        panel.grid.minor   = element_blank(),
        axis.title.y       = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C",
                                          margin   = margin(0, 10, 0, 0)),
        axis.title.x       = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C",
                                          margin   = margin(10, 0, 0, 0)),
        axis.text.y        = element_text(family   = "Lato Full",
                                          face     = "plain",
                                          size     = 3.514598*.pt,
                                          color    = "#524F4C"),
        axis.text.x = element_text(family = "Lato Full",
                                   face   = "plain",
                                   size   = 3.514598*.pt,
                                   color  = "#524F4C"),
        axis.ticks  = element_blank(),
        plot.margin  = unit(c(0, 0, 0, 0), "points")
  )
}
```

Once that you have your theme predefined, you can simply add the newly created `WJP_theme()` function to your ggplot. Take the following example as a guide:

```
# Plot function
  chart <- ggplot(data,
                  aes(x     = grp,
                      y     = perc,
                      label = labels,
                      fill  = column)) +
  geom_bar(stat        = "identity",
           color       = "white",
           show.legend = F,
           position    = "dodge") + #aes(alpha = highlighted)) +
  labs(y = "% of respondents",
       x = "") +
  scale_fill_manual(values  = colors4plot) +
  scale_y_continuous(limits = c(0, 100),
                     labels = c("0%","20%", "40%", "60%", "80%", "100%"),
                     breaks = c(0, 20, 40, 60, 80, 100)) +
  WJP_theme() +
  theme(axis.title.x       = element_blank(),
        axis.title.y       = element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.major.y = element_line(color    = "#D0D1D3",
                                          linetype = "dashed"))
```

3.3 The visualization process

Generating a data visualization might be a complex process. From the conceptual design to the vectorial printing there are many steps involved. Let's focus in the actual programming of the Chart in R. We can divide the visualization code in two stages:

1. Data Wrangling
2. Data Viz

3.3.1 Data wrangling stage

In the **Data Wrangling** stage, the code is focused in preparing the data for the desired visualizations. For example, filtering observations, dropping outliers, grouping observations in predefined groups, estimating certain statistics for each group, among other steps which final outcome is to give a data frame containing *only* the information that is necessary for the visualization.

Given the multiple ways in which the data can be transformed, it is very complex to write a code capable of foreseeing all these possibilities. It is possible to write such a function, but it can quickly turn into a complex task. Therefore, the DAU prefers to automatize this part *only when there is a defined and approved outline*. We do this by writing what we call **Defining Functions**. As a result, each project we work on have their own defining functions.

Usually, a defining function starts like this:

```
figure15.fn <- function() {

  # Variables to plot
  vars4plot = list("Trust"         = c("q1a", "q1b", "q1c"),
                   "Corruption"    = c("q2a", "q2b", "q2c"))

  # Defining data frame for plot
  data2plot <- data_subset.df %>%
    filter(country == mainCountry) %>%
    select(year, all_of(unlist(vars4plot))) %>%
    mutate(
      across(starts_with("Trust"),
             ~if_else(.x == 1 | .x == 2, 1,
                      if_else(!is.na(.x) & .x != 99, 0,
                              NA_real_))),
      across(starts_with("Corruption"),
             ~if_else(.x == 3 | .x == 4, 1,
                      if_else(!is.na(.x) & .x != 99, 0,
                              NA_real_)))
    ) %>%
    group_by(year) %>%
    summarise(across(everything(),
                     mean,
                     na.rm = T)) %>%
    pivot_longer(!year,
                 names_to  = "category",
                 values_to = "value") %>%
    mutate(value = value*100,
           label = paste0(format(round(value, 0),
                                 nsmall = 0),
                          "%")) %>%
    filter(year >= 2014)

  (...)

}
```

As you can see, the main objective of a **Defining Function** is to target a single variable or even a set of variables and apply a cleaning routine that will deliver a data frame ready to be plotted using `ggplot`. We usually refer to this data frame, as `data2plot`. A Defining Function can continue by defining the color palette, the opacity values, and other parameters needed at the **Data Visualization Stage**.

3.3.2 Data visualization stage

Once that we have worked over a `data2plot` and the other parameters needed for visualizing the chart, we call the **Plotting Function**. The main objective of these functions is to receive a data frame such as the `data2plot` and produce a certain type of visualization with it. Therefore, you only need one Plotting Function per chart type (e.g. line, bars, radar, rose, stacked).

Unlike the **Defining Functions**, the **Plotting Functions** just need to anticipate the parameters that can vary between charts such as colors, opacities, shapes, groups, etc. Therefore it is very easy to write general functions that can be reused between projects. The DAU already have a GitHub repository containing most of these **Plotting Functions** which are available to be used for any project that requires similar charts. You can check an example of such function in this page.

> **Important:** These functions are a work in progress by the DAU. Sometimes, it is necessary to add more parameters to the functions to achieve the desired results. In order to avoid the functionality of previous projects, please define these new parameters along with a default value that will refer to the previous functionality of the function. This way, we won't have to change the code from older projects every time that we update the Plotting Functions.