MODULE 6: CREATING INTERACTIVE OUTPUTS & CONCLUSION

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

1. Creating interactive outputs

In the two previous sessions, we learned how to create figures (with `ggplot2`), tables (with, *e.g.* `stargazer`), and documents (with `knitr` and RMarkdown). Those, however, are static outputs: your audience cannot interact with those (*e.g.*, moving the graph, ordering the table, changing the variables, etc.). In some cases, you should give your audience more freedom, so they can better understand your results and findings. R offers multiple tools to allow users to interact with your figures, tables, or data.

1.1 Interactive figures with `plotly`

`plotly` is yet another tool to generate graphs, like `ggplot2`, where the generated graphs will not be an image, but a webpage, *i.e.,* your audience will be able to perform actions on the graph with the mouse in the web browser. Actions include zooming in on the plot, rotating the plot (useful for 3D plots), and selecting observations.

1.1.1 A simple example

- **Warning:** The figures below will not show properly in the pdf, because pdf documents are not interactive. If you want to interact with the figure, run the code in R, or regenerate the document in an HTML output.

```
df <- rio::import("data/graph_reg.Rds")

# Loading plotly
library(plotly)
```

```
Warning: package 'plotly' was built under R version 4.3.1

Loading required package: ggplot2

Warning: package 'ggplot2' was built under R version 4.3.1


Attaching package: 'plotly'

The following object is masked from 'package:ggplot2':

    last_plot

The following object is masked from 'package:stats':

    filter

The following object is masked from 'package:graphics':

    layout
```

- Plotting a simple graph

```
plot_ly(df,
        x = ~iv,
        y = ~dv,
        type="scatter")
```

- Adding a third dimensions

```
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv)
```

- Adding groups

```
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group)
```

- Changing colors of the groups

```
plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group,
        colors = c('#BD382B', '#0C5B8F'))
```

- Changing size

```
# Create a new variable to add as size
df$random_size <- rnorm(nrow(df), 1, 100)

plot_ly(df,
        x = ~iv,
        y = ~iv_2,
        z = ~dv,
        color = ~group,
        colors = c('#BD382B', '#0C5B8F'),
        size = ~random_size)
```

1.1.2 A more complex example: drawing maps

```
# Map example, from https://plotly.com/r/mapbox-county-choropleth/

# Get data
url <- 'https://raw.githubusercontent.com/plotly/datasets/master/geojson-counties-
fips.json'
url2 <- "https://raw.githubusercontent.com/plotly/datasets/master/fips-unemp-16.cs
v"

counties <- rjson::fromJSON(file = url)
df <- read.csv(url2, colClasses = c(fips = "character"))

# Create the map
fig_map <- plot_ly() %>%
  add_trace(
    type = "choroplethmapbox",
    geojson = counties,
    locations = df$fips,
    z = df$unemp
  ) %>%
  layout(
    mapbox = list(
      style = "carto-positron",
      zoom = 2,
      center = list(lon = -95.71,
                            lat =  37.09))
  )
fig_map
```

- `Plotly` is not the only alternative for interactive graphs. Especially, for maps, there is a dedicated tool, called `leaflet`, which is much more handy. An example:
- This was just a very brief introduction to `plotly`. You can read more about it here: this will cover all its possibilities, including other graphs, designs, and interactions you can have with `plotly`.

1.2 Interactive outputs with `shiny`

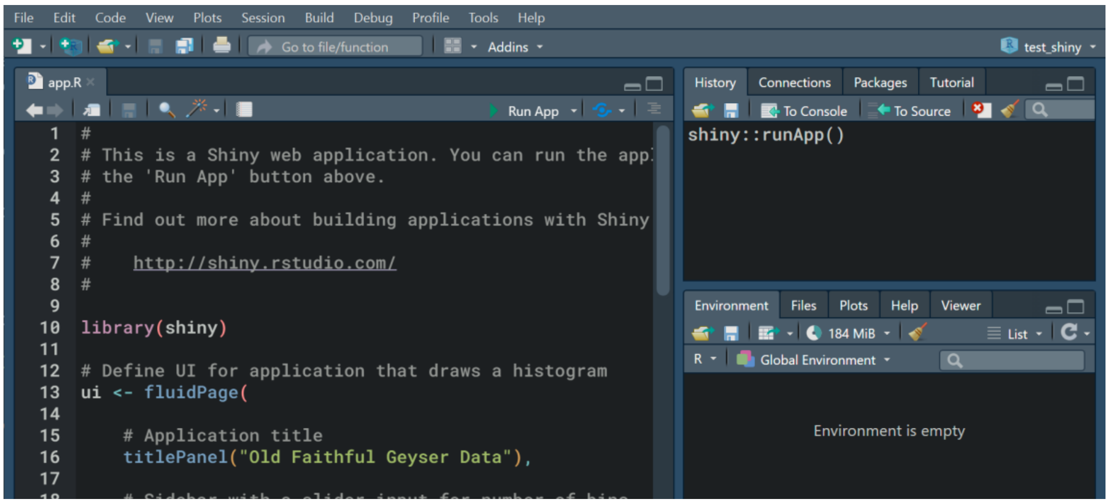
Shiny is a powerful subset of tools, allowing webpages that include buttons, sliders, or text forms. To use Shiny, you must create a specific Shiny project (*i.e.*, this is not a normal R project).

Shiny then contains two elements:

1. A frontend (*i.e.*, the webpage) contains the buttons and the outputs you want in your interactive page.
2. A backend (*i.e.*, a server part) will analyze the request from the webpage and prepare the requested outputs. The R code that you use to type is in the backend. The frontend is mainly composed of HTML markup.

1.2.1 Create a shiny project

- To create a shiny app, go to **File** > **New project...** > **New directory** > **Shiny Application** and give it a name.
- Once you create the new project, a default file called "app.R" appears: this is a shiny app you can run.



```
#
# This is a Shiny web application. You can run the app
# the 'Run App' button above.
#
# Find out more about building applications with Shiny
#
#    http://shiny.rstudio.com/
#

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

```

```
shiny::runApp()
```

Environment is empty

The shiny app

- To run the shiny app, click `Run app`. It will start a server, and open the app in the web browser.

1.2.2 The Shiny UI

- The shiny app contains two parts. The first part is a `ui` function (_i.e. User Interface):

```
# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)
```

- The function contains the elements that you want to display on the page. The app above contains:

1. A title, created with `titlePanel()`
2. A slider input, created with `sliderInput()`
3. A plot output, created with `plotOutput()`

- The you see that the slider and the plot are respectively placed in a `sidebarPanel()` and a `mainPanel()`. If you run the shiny app, you will see the elements placed on your screen.
- Shiny apps can contains many other elements, but they are always placed on the screen using the same logic: invoking the name of the object with a function.
- The list of objects you can place are available here: https://shiny.rstudio.com/reference/shiny/1.6.0/

1.2.3 The Shiny Server

- The second important part of a shiny app is the **server**. This is what gets the inputs (such as the `sliderInput()` above) and generate the outputs (such as the `plotOutput()` above).
- The server side is also a function:

```
# Define server logic required to draw a histogram
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # Here, you get the data that you want to display:
        #   this could come from a data file.
        x    <- faithful[, 2]
        # Here, getting the input from the UI slider, you create
        #   a bin variable
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        #  --> This is the thing your renderPlot function returns
        hist(x, breaks = bins, col = 'darkgray', border = 'white')
    })
}
```

- The function contains two parameters: `input` and `output`. They are the two lists of inputs and outputs defined in the UI. For example, in the UI, we created a `sliderInput()` called `"bins"`: it is accessible from the server side, as `input$bins`. In this case, as `bins` is a slider set to a value between 1 and 50, `input$bins` is a value between 1 and 50. Every time the slider is moved by the user, the value of `input$bins` is updated. This value can be used in any R code.
- The other parameter is `output`. This one expects the element to render on the webpage. For example, the UI contains a plot called `"distPlot"` (see the UI code, above). This means that the server needs to render a plot and save it in the list of outputs, in an object called `distPlot`: then, the UI can access it. This is what we have in the code, with the function `renderPlot({})`, assigned to `output$distPlot`.
- Every time the user interface is updated (for example, the slider is moved), the functions of the servers are rerun, and the new plot is rendered.

1.2.4 Deploy the app

- The app that you have designed is run locally, on your computer. This can be handy during presentations sometimes, but this is not enough to let other people interact with it.
- If you want to share it online, deploying it on a web server is possible. This requires some skills in web deployment. However, you can also find ready-made cloud solutions to deploy your app. For example, RStudio allows you to deploy your app (*i.e.,* make it accessible through the web browser of other people) on their website, here: https://www.shinyapps.io/
- Shinyapps.io is not free for large-scale applications, but you can test it and develop personal projects with the free version.

1.2.5 Read more about `shiny`...

- The shiny developers created a complete course accessible for free. If you would like to learn more about it: https://shiny.rstudio.com/tutorial/

2. Final words, don't forget style and elegance

So far, we used short and easy examples. If you plan to continue using R, either for your thesis or in your job, you will soon end up with more complex datasets and projects. In this final section, I give you some advice to improve the speed and usability of your code:

2.1 Advice on speed

2.1.1 Use vectors!

R is a vector language. Manipulate vectors, use vectors. To the contrary

2.1.2 Don't use loops!

In R, loops are very, **VERY** inefficient! If you have one million observations, it might take hours to run your loop. This does not work. R is designed to manipulate vectors. As much as you can, manipulate vectors. Examples: see module 5!

2.1.3 data.frame are slow!

- Accessing a `data.frame` takes time for R, especially if the `data.frame` contains hundreds of thousands of variables.
- If you have to access a data.frame multiple times for specific variables (for example, in a loop that you cannot avoid), it is faster to extract the variable from the data.frame, compute it, and then save it back:
- **Example:** This, is faster (and gets even faster compared to the other code below if `df` contains many variables):

```
var_1 <- df$var_1
for (i in 1:length(var_1)) {
  var_1[i] <- var_1[i] + 2
}
df$var_1 <- var_1
```

- than this:

```
for (i in 1:length(var_1)) {
  df$var_1[i] <- df$var_1[i] + 2
}
```

- **Note** that both codes above make no sense: you should use vectors to compute var_1 in both cases.

2.2 Advice on memory

2.2.1 Mind you data types!

- If your computer has memory issues, you may want to reduce the size of objects. Not all objects in R are made equal! A `logical` takes much less space than a `character`. For example

```
# A dummy variable with 200000 observations, NOT properly coded
student_level <- c(rep("undergrad.", 100000), rep("postgrad.", 100000))
object.size(student_level)

1600176 bytes

# Now, the same information, as a logical variable
is_undergrad <- c(student_level == "undergrad.")
object.size(is_undergrad)

800048 bytes
```

- You will note that the `is_undergrad` object (logical) above is twice smaller than the `student_level` object (character) although they contain the same information.
- In general:
  - Using `factor` is cheaper in memory than using `characters`
  - Using `integer` (if possible) is cheaper than using `numeric`
  - `logical` is the cheapest (if possible)

2.2.1.1 Clean after you

- If you stored and created many objects in your environment, you may run out of memory. To get free space again, delete the objects and free the memory.
- To delete an object

```
rm(tmp)
Warning in rm(tmp): object 'tmp' not found
```

- To collect garbage (= free memory that is not used anymore)

```
gc(reset = TRUE)

          used (Mb) gc trigger  (Mb) max used (Mb)
Ncells  983962 52.6    1970208 105.3   983962 52.6
Vcells 1945685 14.9    8388608  64.0  1945685 14.9
```

2.3 Advice on code

2.3.1 Dummy variable: be affirmative!

- It is often the case, when you see a dummy variable in dataset, that it is coded as 0, 1, with an obscure name (for example, `student_educ_level`) and a label indicating the levels (*e.g.* `bachelor`, `master`...). This is not ideal, because the variable name does not indicate what coding corresponds to 0 or 1.
- You should use affirmative statements in the dummy variable names instead. For example, `is_bachelor_student` is better than a dummy `student_educ_level` variable, because the statement `is_bachelor_student` can be `TRUE` or `FALSE`. On the contrary, student education level cannot be `TRUE` or `FALSE`, hence not conveying the correct information.

2.3.2 Use functions!

It is very often that you want to proceed to multiple steps in your analyses. The usual practice is to write hundreds of lines of code in a row. This is a bad practice because it makes tracking changes in your variables very difficult.

- If you want to perform any operation, such as computing `var_3`, create a function for this:

```
compute_var3 <- function(var_1, var_2) {
  # Your code
}
```

- By doing so, you know exactly where is all the code related to the creation of your var_3 variable. You can easily run it independently and store it in a different file.

2.3.3 Separate data manipulation from data analyses!

- In general, when you analyze data, you should store the preparation of the dataset and the analysis in different files. This avoids having multiple versions of the same variable, used in different analyses (for example, if you run a linear model with `var_3`, then make changes on `var_3`, then run another linear model, you will not be able to tell that `var_3` changed in the meantime by looking at the output).
- In any data analyses process, you should follow the steps:

1. Load the raw data
2. Process the data with necessary steps (remove missing, compute new variables, etc.)
3. **Save the complete data manipulation in a new file, in `.Rds` format**
    1. Do not forget that other formats may be lossy: information from data.frame may not be recorded properly in Stata files, Excel files, or other file formats.
    2. If data are not anonymized, you should always prefer to anonymize or pseudonymize them before saving your cleaned data file. In case of pseudonymization, you should store the identification keys separately from the data.
    3. REMEMBER: NEVER UPLOAD A DATA FILE ON GITHUB!
4. Only use the cleaned file in `.Rds` format when creating your outputs and analyses: you should not change anything to your data anymore at this stage to make sure everything is transparent and reproducible.
    1. Save ALL your analyses in R scripts or Quarto files so that you can reproduce them later if required.

3. Conclusion

This was only a short introduction to R. As in every language, the best way to move further is to start speaking... or writing. And don't forget, if there is something you don't know how to do it yet, the answer is probably on StackOverflow.