PSYC166 Exercise: Functions, Scripts, and Reports

# Scripts Containing Instructions

You can write code for a project in a single file. That file can contain instructions for reading, cleaning, manipulating, plotting, and modeling data as long as those tasks are coded linearly. A downside of such an approach is that tasks are not compartmentalized. When collaborating, all collaborators work in a single file, rather than by working on parts of the project. By way of analogy, a building construction company would have all workers working in one huge open room without any physical barriers like walls or floors. The metal machining team would work some feet away from the marketing team and the accounting team. Sound and debris would be cast throughout the structure. Sure, everyone is in close proximity thus facilitating communication across departments from start to finish but such a structure would make little sense.

Consider the problems associated with coding in the same way. When code breaks (which it does for different reasons), the entire file (up to a point and beyond) will discontinue working. Finding and fixing the errors in a single file may require a lot of searching and editing, which may inadvertently result in changing some other code, if even by accident. Keeping tasks separate will reduce the likelihood of accidentally introducing errors. When coding, a task should be compartmentalized from other tasks serving different functions and goals.

For your project, you will read, manipulate, and write data. The code containing the instructions to read, manipulate, and write data will be contained within two types of files, Scripts (e.g., .R) and RMarkdown files (e.g., .Rmd). The focus those files will be on reproducibility. In order to have reproducibility, you will need working code as well as code that executes without you doing so manually. In order words, you will work with code files that when executed will call and execute other code files so that your objects, files, visualizations, and models will update automatically.

You can think of the structure as a building of floors. Each floor rests on the infrastructure of all lower floors. In order to reach the top floor, you pass through each floor, whether manually (walking stairs) or automatically (riding the elevator).

- etc.  
  
- File\_03 - code loads libraries/functions,   
 calls File\_02, reads transformed data, cleans/adds variables  
  
- File\_02 - code loads libraries/functions,   
 calls File\_01, reads subset file, transforms data subset,   
 writes out transformed data   
  
- File\_01 - code loads libraries/functions, reads data, subsets data,   
 writes out data subset

Scripts will create, change, and remove objects based on your code instructions. The first step will be to create a script.

# Creating and Editing Scripts

## Creating a Script

Create another .R script file:

1. Name the file ex02.R
2. Save to the project directory where you would save your script files

### Editing a Script

Inside ex02.R:

1. Create a vector object containing 5 birth dates according to format "yyyy-mm-dd" and assign it a meaningful name.
2. Create a vector object containing 5 names of individuals and assign it a meaningful name.
3. Save your script file.

**Recommendation:** Consider adding comments about what your code instructions are doing so that you can be reminded easily.

## Creating a Function Script

Most of the functions you will use will in libraries that you will load. Other libraries, which you obtain someplace or write yourself, may be inside specific files. You will now create a function script.

Create another .R script file:

1. Name the file get\_years\_since\_birth.R
2. Save to the appropriate project directory

## Editing a Function Script

Inside get\_years\_since\_birth.R, add the function definition.

get\_years\_since\_birth <- function(dob) {  
   
 # make string a data  
 dob = lubridate::as\_date(dob)   
   
 # obtain the difference in time in days  
 diff = difftime(time1 = Sys.Date(), time2 = dob, units = "days")  
   
 # create age base on days  
 age = as.numeric(diff / 365.25)  
   
 # return the age in years, truncated   
 return(trunc(age))  
}

## Running the Code Definition

Once you write a function, you would want to ensure it works correctly. In order to do so, you would execute the code inside that function file. We will skip this step because a) the function works and b) we will load the definition by sourcing the script function.

The next step will be to edit the script to create new objects based on some functions.

## Editing a Script (again)

You will certainly be making change to your code and hopefully saving your work. You will also likely work with more than one file, which you will have to remember to edit and save, especially when the code instructions in one file depend on another file. If the new instructions in one file are not saved, another file will not assume those changes. Let’s got back and edit that other file.

Inside ex02.R:

1. Create a new vector object the contains the ages of the individuals in the birth dates vector by using the get\_years\_since\_birth() function and name the object in a meaningful way.
2. Create a new object that contains the mean of this new vector object. Use the base R function, mean() and name the object mean\_age.
3. Save your script file.

The next step will involve automating the function definition in your script by sourcing the file containing the function or directory containing function files. Like loading libraries, this step allows you to ensure the functions are available in your workflow.

# Sourcing Script Files

After creating script files, whether code files or function files, you will likely need them executed so that you can fulfill tasks that depend on them. You want to automate this task so that you do not have to execute your scripts manually. The processing of calling a code file inside another file is referred to as ***sourcing***. We will source the function file for creating objects in ex02.R.

## Sourcing A Function File

You now have a function that you might use in performing some tasks. This function serves as an example of how to deal with any personal functions that you create and will need to use for your project. You need to ensure that the function is both available and also defined (executed) so that you can use it to tackle a task. You want to automate this process so you don’t need to do so manually.

1. At the top of ex02.R, near where you load your libraries, add a comment to describe what you are doing with this code. your future self will thank you.
2. Below that comment, add code to source your function file

source r/functions/get\_years\_since\_birth.R file specifically

1. Keep in mind that the first argument you need to pass in source() if the file, which help(source) will tell you is the *‘string giving the pathname’*. You will need to ensure your string is the path to the file.

* rather than hard code the path, use the **{here}** library to help you locate the file. See examples in class module.

## Sourcing Function Files from a Directory

Assuming you save any personal functions in r/functions, you may wish to source all functions saved there rather than source specific ones.

1. Add a comment to describe what you are doing with this code.
2. The function you can use for talking this tasks is from **{R.utils}**:

R.utils::sourceDirectory()

This function will look for files in the directory you specify and will source them all.

1. Source all files in the r/functions directory. Again, rather than hard code the path to the *directory*, use **{here}** to point to that directory path.

**NOTE:** Remember, this is a directory and not a file.

1. You can comment out your code from above and add new code so that only one of the source calls will run in your code.
2. If you see other functions from class or otherwise, you can create a file and add the function to the directory.

The next step involves creating an R Markdown report file that incorporates the other work and leverages it for reproduction.

# Creating Reports

Report files will be created using R Markdown and saved in /report. Report files will source code script files, which will in turn source function files. In order to build clean reports, you need to create a mental model for understanding how files leverage the contents of other files and for organizing them.

- R Markdown for written text,   
 loads any libraries  
 sources script files (e.g., ex02.R)  
 utilized objects for text   
  
- R script file for task,   
 sources function files (e.g, function\_name.R),   
  
- function file, function\_name.R

## Creating the R Markdown Report File

Create an .Rmd R Markdown file:

1. Name the file ex02\_report.Rmd
2. Save to the appropriate project directory, which should be /report.
3. Clean the unnecessary content from RStudio’s starter R Markdown file.
4. Create a code block, within which you source your ex02.R file. To ensure portability, however, add the following argument the local parameter of source():

* source(file = path\_to\_file, local = knitr::knit\_global())

1. Set the code block option to “show nothing (run code)”.
2. After your code block, add text that states “The mean age of the individuals is mean\_value”. Importantly, rather than hard-coding this value, use in-line code to call the r object that’s created in ex02.R which holds this value.

If you have forgotten how to render R objects in-line, you would be with a backtick and lowercase r to invoke the R interpreter.

Example:

The mean age of the individuals is `r mean\_age`.

1. Knit your .Rmd file to render it as an HTML file.
2. Knit your .Rmd file to render it as a MS Word file.

# Summary

You have now created a very simple report. Importantly, you have assembled all of the pience for a simple report without embedding them into a single file. In order to produce the report, you created the necessary basic files and have organized those files in a way to build the report.

You have:

1. created a function file containing a function;
2. created a script file containing numeric and character vectors and applied functions to compute new objects;
3. sourced functions and scripts leveraging the **{here}** library;
4. created a markdown file containing code blocks and in-line r code which is reproducible and automated

Although your team project will contain more parts and files (e.g., libraries, data frames, reading data, writing data, summarized data frames, plots, model objects, etc.), you have worked through the mental model for organizing a project and its essential components. Building reports this way allows for your work to be reproduced easily when you need to update data, update models, update plots, etc.

# Working Smarter, Not harder

## Using Code Snippets

You can work hard or you can work by being smart. When you have code that you may use often, you can turn them into code snippets to reuse without needed to retype fully.

1. Navigate to **Tools > Edit Code Snippets**
2. At the bottom of your snippet file, add code like:

snippet source\_dir  
 R.utils::sourceDirectory(here::here("r", "functions"))

1. Save the snippet file
2. Now, within the R console or in your coding environment, you can type sour and the RStudio intellisense will populate the snippet name, source\_dir containing the code. Pressing return/enter will add the line of code to your environment. Test when ready.