Organizing

EDH7916 | Spring 2020

Benjamin Skinner

Organizing a project directory

In this lesson, we'll discuss how to organize both a project directory and an R script. While there's no one exact way to do either, there are good practices that you should generally follow. We'll begin with how to organize your project files.

A place for everything and everything in its place

Every data analysis project should have its own set of organized folders. Just like you might organize a kitchen so that ingredients, cookbooks, and prepared food all have a specific cabinet or shelf, so too should you organize your project.

But computers are pretty good at finding files, you say: you can use your machine's search feature look for what you need. If you don't have that many files to look through, you might not be too bad at quickly scanning to find what you want either. If this is the case, then why bother organizing a project directory? Why not just dump everything — scripts, data, figures, tables, notes, *etc* — into a single folder (*My downloads folder works just fine, thank you.*)? If you need something, the computer can definitely find it.

So what's the big deal?

The big deal is that you are thinking from your computer's perspective when you should be thinking from the perspective of you, your collaborators (which includes your future self), and future replicators (which also includes yourself). Search features are nice, but there's no substitute for being able to look through a project's files *just by looking through the project folders*. When a project is well organized, it's much easier to understand how it all fits together: the inputs, processes, and outputs.

A common directory structure

You already have a basic directory structure with your class repo:

```
student_<your_last_name>/
|
|__ assignments/
|__ data/
|__ lessons/
|__ scripts/
|__ working/
```

With a project, you probably won't have `assignments` or `lessons` folders, but you almost certainly will have folders for your `scripts` and `data` as well as a `working` folder (which some people call `scratch`, like a scratch pad).

What are we missing? Following our analogy above, we have:

- **Ingredients (Inputs)** - `data`
- **Cookbooks (Processes)** - `scripts`

We are missing **Prepared food (Outputs)**!

What kinds of outputs are we likely to have with our project? One type quickly come to mind: figures. Both when we start exploring our data and when we want to make a final report to share, we're likely to make a number of plots. Let's make a folder.

> **Quick exercise** Make a folder in your `student_*` working directory called `figures`. There are two ways to do this: (1) you can use the point-and-click interface in the bottom right panel of RStudio, or (2) you can use the function `dir.create()` in the console.

Absolute vs relative paths

Another reason we organize our project directory is so that we have a common framework for loading, analyzing, and saving our work. This common framework allows us to use **relative** rather than **absolute** paths in our scripts.

A quick analogy. Imagine you ask a friend for directions: *how do I get to the state park?* Your friend has three ways they can give you directions:

1. How to get to the park from *their* house
2. How to get to the park from *your* house
3. How to get to the park from a common landmark

The first way is really easy for your friend, but it puts extra work on you. Either you first must go to their house so that the directions make sense, or you have to translate them from your perspective. These directions are annoying and useless if you also don't know how to get to their house.

The second way is really nice to you, but difficult for your friend. These directions also aren't reusable for other people. If a third friend tries to get the directions from you, you'll just replicate the first problem. Alternately, if you aren't at your house the directions aren't as useful.

The third way is the best compromise. If you know how to get to a common landmark, you can get there the way that is best for you. Once there, you can follow your friend's directions to the park. These directions are also shareable: anyone who can get to the common landmark can use the directions to get to the park.

In the world of your project, the first two sets of directions are **absolute** paths while the third is a **relative** path. If your project is organized into folders and another person (collaborator, code reviewer, future you on another computer) has the full project directory, then as long as they can get to the correct starting point — *e.g.* your `scripts` folder — they can run your project easily.

Examples

- **Absolute path**: `/Users/btskinner/classes/edh7916/scripts`
- **Relative path**: `./scripts`

The second line assumes we are in the class directory (your `student_*` folders). Notice the `.` (dot)? On almost all systems, dots in a file path work like this:

- `.` (one dot): *this* directory/folder
- `..` (two dots): *up one* directory

What if we are in the `scripts` folder, but want to access a file in the `data` directory? We use two dots (`..`):

- from `scripts` to `data`: `../data`

What this says is

1. `..`: back up into the main folder, `student_*`
2. `/data`: from the main folder, go into `data`

Quick exercise Use R's `list.files()` command in the console to show all the files in the current directory. Next, use the same command, but use a relative link to see all the files inside the `scripts` folder. Finally, use a relative link to see all the files in the directory above.

Quick exercise Create a file in you new `figures` directory called `README.md`. While you can use RStudio's point-and-click interface, I want you to use the R command `file.create()`. Be sure to include the proper path so that it's created in the `figures` folder.

Organizing a script

Do one thing and do it well

A general philosophy for every script (and functions within a script, for that matter) should be to do one thing and do it well. Rather than have a single giant script with 5,000 lines that does your entire project, it makes more sense to break up your scripts into shorter modules that you can then bring together. Right now, we have small scripts so having just one is fine. But this idea applies within the script itself: organize your scripts into clear sections so that a person can easily scan it to see what is happening and where.

Template

Here's a template for an R script with clearly defined sections:

```
###############################################################################
##
## [ PROJ ] < Name of the overall project >
## [ FILE ] < Name of this particular file >
## [ AUTH ] < Your name + email / Twitter / GitHub handle >
## [ INIT ] < Date you started the file >
##
###############################################################################

## ---------------------------
## libraries
## ---------------------------

## ---------------------------
## directory paths
## ---------------------------

## ---------------------------
## settings/macros
## ---------------------------

## ---------------------------
## functions
## ---------------------------

## -----------------------------------------------------------------------------
## < BODY >
## -----------------------------------------------------------------------------
```

```
## ---------------------------
## input
## ---------------------------

## ---------------------------
## process
## ---------------------------

## ---------------------------
## output
## ---------------------------

## -----------------------------------------------------------------------------
## END SCRIPT
## -----------------------------------------------------------------------------
```

And here's the template with a very simple project outline:

```
################################################################################
##
## [ PROJ ] EDH 7916: Organizing
## [ FILE ] organizing.R
## [ AUTH ] Benjamin Skinner (@btskinner)
## [ INIT ] 13 January 2020
##
################################################################################

## ---------------------------
## libraries
## ---------------------------

library(tidyverse)

## ---------------------------
## directory paths
## ---------------------------

dat_dir <- file.path(".", "data")
fig_dir <- file.path(".", "figures")

## ---------------------------
## settings/macros
## ---------------------------

old_to_new_score_ratio <- 1.1

## ---------------------------
## functions
## ---------------------------

old_to_new_score <- function(test_score, ratio) {
    return(test_score * ratio)
}
```

```
## ------------------------------------------------------------------------
## BODY
## ------------------------------------------------------------------------

## ---------------------------
## input
## ---------------------------

df <- readRDS(file.path(dat_dir, "test_scores.RDS"))

## ---------------------------
## process
## ---------------------------

## add a column for new test score
## <dataframe>$<new_scores_column> <- function(<old_scores_column>, <ratio>)
df$test_scores_new <- old_to_new_score(df$test_score, old_to_new_score_ratio)

## ---------------------------
## output
## ---------------------------

saveRDS(df, file.path(dat_dir, "test_scores_updated.RDS"))

## ------------------------------------------------------------------------
## END SCRIPT
## ------------------------------------------------------------------------
```

Header

At the very top of your script, give all the relevant information about the script.

```
###############################################################################
##
## [ PROJ ] EDH 7916: Organizing
## [ FILE ] organizing.R
## [ AUTH ] Benjamin Skinner (@btskinner)
## [ INIT ] 13 January 2020
##
###############################################################################
```

Specifically:

- `[ PROJ ]`: tell what project it belongs to
- `[ FILE ]`: give the file's name
- `[ AUTH ]`: give your name and a way to contact you
- `[ INIT ]`: give the date you started the file

If you aren't using a version control system like git, then it would make sense to also include a line for the last time you revised the file: `[ REVN ]`. You can still do this if you want, but the date you started the file in conjuction with the changes you commit to git should suffice to give its history.

NOTE You don't have to make your header look *exactly* like mine. This is just what I've landed on after a few years. As long as you have the relevant information, personalize the details as you will.

Libraries: what extra code do we need to make this code work? After the informational header, the first thing you want to include are the libraries you need to call for your script to work. In this course, you will almost always call the **{tidyverse}** library.

```
## ---------------------------
## libraries
## ---------------------------

library(tidyverse)
```

Paths: where is everything and where is it going? Notice our relative links? Rather than hard-coding / rewriting all the paths in the script below, we can save the paths in an object. We use the `file.path()` command because it is smart. Some computer operating systems use forward slashes, `/`, for their file paths; others use backslashes, `\`. Rather than try to guess or assume what operating system future users will use, we can use R's function, `file.path()`, to check the current operating system and build the paths correctly for us.

```
## ---------------------------
## directory paths
## ---------------------------

dat_dir <- file.path("..", "data")
fig_dir <- file.path("..", "figures")
```

> **Quick exercise** Run these lines of code and then print `dat_dir` and `fig_dir` to the console. What do you see?

Settings/Macros: what numbers, options, or settings should be consistent? Programmers hate "magic" numbers. What are "magic" numbers (including "magic" strings and settings here as well)? They are values that are hard-coded in your analysis script.

Say you want to convert old test scores to a new test score based some ratio, let's say 1.1 (10% increase). You could multiply every old test score value throughout your script by 1.1, but what if you later decide it should be 1.2 or 0.9? You need to change every instance of 1.1. (Don't miss any!) Alternately, does 1.1 have any inherent meaning? To me, not really.

```
## ---------------------------
## settings/macros
## ---------------------------

old_to_new_score_ratio <- 1.1
```

It's better to store these constant reusable values at the top of your script in an object (or macro, as I call it here) that has a clear name. `old_to_new_score_ratio <- 1.1` is clear in its meaning, so when we use it below, we'll know what it means. Also, if we decide we need to fix / change the value later, we only have to change it once.

Put all such numbers, strings, and settings here.

Functions: what useful code will be repeated below? Other than the functions that come with the packages we load, we might write functions ourselves. At this point in the course, I'm not concerned that you know how to write a function or how one works. Just notice how the function has a good name that tells what it does: converts old scores to new scores using a ratio (we'll use the one we defined above).

```
## ----------------------------
## functions
## ----------------------------

old_to_new_score <- function(test_score, ratio) {
    return(test_score * ratio)
}
```

Body

The body of your script is where the main work happens. You may need more sections, but at the very least you should generally have dedicated spots for reading in your data, working with your data, and saving any output.

Input: read in (or create) data we'll work with Here we read in a very small tibble (a tidyverse version of a data frame) with some student IDs and their test scores. We can see this if we look at the data.

```
## ----------------------------
## input
## ----------------------------

df <- readRDS(file.path(dat_dir, "test_scores.RDS"))
```

> **Quick exercise** Look at the data, both using RStudio's **View** and by using some of the basic techniques we've learned so far.

Process: do the analytic work Let's say that all we want to do is add a new column in which we have the new test scores as computed with our function and ratio. We can use base R to add a new column using the `$` notation.

```
## ----------------------------
## process
## ----------------------------

## add a column for new test score
## <dataframe>$<new_scores_column> <- function(<old_scores_column>, <ratio>)
df$test_scores_new <- old_to_new_score(df$test_score, old_to_new_score_ratio)
```

Output: write the results from our analyses Finally, we save our tibble. Notice how we use a new name. We'll talk more about data consistency in a couple of lessons, but we never want to overwrite our original data. Save a new file with a useful name to keep everything separate.

```
## ----------------------------
## output
## ----------------------------

saveRDS(df, file.path(dat_dir, "test_scores_updated.RDS"))
```

Two considerations about naming files and macros

When naming folders, files, objects, or macros, keep these naming rules in mind:

1. Name it well (be clear): *e.g.* `data_clean.R` or `old_to_new_score_ratio`

2. Name it consistent with other items (have a style): *e.g.* `data_1.RDS`, `data_2.RDS`,`data_3.RDS`, *etc*
3. Name it without spaces!
    - **NO**: `data clean.R`
    - **YES**: `data_clean.R`

More person hours are lost than can be counted dealing with file names with spaces. Don't do it! Use underscores or hyphens to separate words.