Making Packages in R (Summary)

Here is a summary of all the steps we took to make an R package during 2023-03-03 lab meeting. Note that all of this information and much much more can be found in extreme detail here: https://r-pkgs.org/

The task

We are given a .csv file containing mouse group (mTBI or sham), axon diameters, and fiber diameters. Our goal is to do a t-test comparing the g-ratios of the mTBI mice with those of the shams.

Achieving our goal in a regular R script

```
library(here)
```

The `here` package guesses your project's "root" directory is and uses that as the working directory. This is one way to avoid manually specifying your working directory with `setwd()`.

```
# Load the data
mouse_em <- read.csv(here("data/mouse_em.csv"))

# Take a look at the first few rows
head(mouse_em)
```

```
##   axon_diameter fiber_diameter group
## 1     1.3709584      0.7368337  mtbi
## 2     0.5646982      1.7940778  mtbi
## 3     0.3631284      3.6065798  mtbi
## 4     0.6328626      1.4714902  mtbi
## 5     0.4042683      0.2054473  mtbi
## 6     0.1061245      0.6329789  mtbi
```

```
# Calculate the g-ratios
mouse_em$g_ratio <- mouse_em$axon_diameter / mouse_em$fiber_diameter

# Take a new look at the first few rows
head(mouse_em)
```

```
##   axon_diameter fiber_diameter group   g_ratio
## 1     1.3709584      0.7368337  mtbi 1.8606076
## 2     0.5646982      1.7940778  mtbi 0.3147568
## 3     0.3631284      3.6065798  mtbi 0.1006850
## 4     0.6328626      1.4714902  mtbi 0.4300828
## 5     0.4042683      0.2054473  mtbi 1.9677472
## 6     0.1061245      0.6329789  mtbi 0.1676588
```

```
mtbi_g_ratios <- mouse_em$g_ratio[mouse_em$group == "mtbi"]
sham_g_ratios <- mouse_em$g_ratio[mouse_em$group == "sham"]

t.test(mtbi_g_ratios, sham_g_ratios)$p.value
```

```
## [1] 0.04999999
```

Success! Let's look at the code we actually ran without the clutter:

```
mouse_em$g_ratio <- mouse_em$axon_diameter / mouse_em$fiber_diameter
mtbi_g_ratios <- mouse_em$g_ratio[mouse_em$group == "mtbi"]
sham_g_ratios <- mouse_em$g_ratio[mouse_em$group == "sham"]
t.test(mtbi_g_ratios, sham_g_ratios)$p.value
```

It is only 4 lines here, but you can imagine that for a different analysis, the code could have become quite long.

Making the R package

To make this into a package, we are going to do the following steps:

Brief overview

1. Run `library(devtools)` in the *console* (considered better practice than running it from the script)
2. Run `create_package("path/to/our/package")` to initialize our package at a path of our choice
3. Create a file to store our code in the `R` directory of our package.
4. Paste our script code into this file and reformat to make it more package-friendly
5. Document our function appropriately
6. Specify a license (`use_mit_license()` in the console)
7. Run `check()` in the console to make sure our package has no issues
8. Run `install()` in the console to install our package!

In detail

Run the following lines of code in the console, making sure to replace the "path/to/your/package" with an actual path you want to store your packages source code at.

I would suggest that you do **not** store your main copy of your package on a automatic-syncing cloud storage system like onedrive or googledrive. Writing directly to a mounted network drive like carbon (or onedrive/googledrive if you have somehow mounted those) is fine. The reason for this is that onedrive and google drive seem to enjoy guessing what to do when conflicting files are present across your devices, and their best guess might corrupt your package at some point or another. Having a local copy of your package on your system and manually backing it up is the second best thing you can do, with the best being a github repository (but that is for another workshop).

```
library(devtools)
create_package("path/to/your/package")
```

If you are using RStudio, a new session of RStudio should appear. This session has your package location set as the working directory. Unless specified otherwise, we will do everything from now on in this new session.

The next step is to create a file where you will store your analyses. R packages are mostly just a collection of functions (see end of this document for more information). We could keep all of our functions in a single R file, but it is often neater to group similar functions together in their own file.

Create a new .R file called "ttests.R" and save it to the R folder within your package. You can alternatively type `use_r("ttests")` in the console.

In this R file, paste in the code we put together earlier:

```
mouse_em$g_ratio <- mouse_em$axon_diameter / mouse_em$fiber_diameter
mtbi_g_ratios <- mouse_em$g_ratio[mouse_em$group == "mtbi"]
sham_g_ratios <- mouse_em$g_ratio[mouse_em$group == "sham"]
t.test(mtbi_g_ratios, sham_g_ratios)$p.value
```

Now let's wrap this in a function:

```
g_ratio_t_test <- function(mouse_em) {
    mouse_em$g_ratio <- mouse_em$axon_diameter / mouse_em$fiber_diameter
    mtbi_g_ratios <- mouse_em$g_ratio[mouse_em$group == "mtbi"]
    sham_g_ratios <- mouse_em$g_ratio[mouse_em$group == "sham"]
    return(t.test(mtbi_g_ratios, sham_g_ratios)$p.value)
}
```

If we leave it like this, the R package builder won't be happy. Any variables (stuff not in strings or constants) are not going to be allowed into the party unless the builder knows where they came from. To be honest, that description may be very inaccurate to what is going on behind the scenes, but it gets the job done.

Our code has a data object `mouse_em` which we introduce as a parameter for the function, but things like `axon_diameter` and `t.test` are complete mysteries to whatever forces control R package building. There are multiple ways to work around this issue, but one option is to wrap non-parameter variable in quotation marks:

```
g_ratio_t_test <- function(mouse_em) {
    mouse_em$"g_ratio" <- mouse_em$"axon_diameter" / mouse_em$"fiber_diameter"
    mtbi_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "mtbi"]
    sham_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "sham"]
    return(t.test(mtbi_g_ratios, sham_g_ratios)$"p.value")
}
```

The `t.test` function needs some special treatment. When using a function that is provided by another package, we need to explicitly mention which package gave us that function in our code. We also need to specify that this is a package we need our package to have access to when it is being built.

To do the former, specify that t.test comes from the `stats` package:

```
g_ratio_t_test <- function(mouse_em) {
    mouse_em$"g_ratio" <- mouse_em$"axon_diameter" / mouse_em$"fiber_diameter"
    mtbi_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "mtbi"]
    sham_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "sham"]
    return(stats::t.test(mtbi_g_ratios, sham_g_ratios)$"p.value")
}
```

And to do the latter, run the following line in the console:

```
use_package("stats")
```

The last thing we need to do is document our function appropriately:

```
#' Calculate t-test between mtbi and sham g-ratios
#'
#' @param mouse_em A dataframe containing axon_diameter and fiber_diameter vars
#'
#' @return pval The p-value of the t-test
#'
#' @export
g_ratio_t_test <- function(mouse_em) {
    mouse_em$"g_ratio" <- mouse_em$"axon_diameter" / mouse_em$"fiber_diameter"
    mtbi_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "mtbi"]
    sham_g_ratios <- mouse_em$"g_ratio"[mouse_em$"group" == "sham"]
    pval <- stats::t.test(mtbi_g_ratios, sham_g_ratios)$"p.value"
    return(pval)
}
```

The documentation lines provide a description of the function, the parameters, the returned object, and finally an `@export` line to indicate that we want users who load our package to instantly have access to this function. You can learn more about this documentation style by looking into Roxygen2.

Next, we need to specify a license. Run in the console:

```
use_mit_license()
```

Finally, we can check to make sure everything is functioning properly and install the package if there are no warnings or errors. In the console, run:

```
check()
```

Followed by:

```
install()
```

Now wherever we are on our computer, we can always get that p-value with our function:

```
library("yourpackagename")

mouse_em <- read.csv("data/mouse_em.csv")

pval <- g_ratio_t_test(mouse_em)
```

A workflow that isn't a huge pain

1. Try to make your code in function form right out of the gates
2. Workshop your functions by executing them from your main script
3. When they're good to go, throw them in your package files and document appropriately
4. Use `check()` regularly to make sure disaster hasn't struck

Functions

In case the concept of functions is a little fuzzy, here is a short overview. Functions are code "things" that execute a particular chunk of code given some provided parameters.

```
# Non-function way
mass <- 5
acceleration <- 3
force <- mass * acceleration
print(force)
```

```
## [1] 15
```

```
# Function way
calculate_force <- function(mass, acceleration) {
    force <- mass * acceleration
    return(force)
}

calculate_force(mass, acceleration)
```

```
## [1] 15
```

The function looks more complicated here, but the important thing is that you only need to define a function once, making it much easier to re-run the same code for different starting values:

```
calculate_force(3, 4)
```

```
## [1] 12
```

Despite hiding the raw source code away from the user, a properly named function can make the overall code much easier to read. Seeing one line that says `t.test` is much easier to digest than the monstrosity happening behind the scenes:

https://github.com/SurajGupta/r-source/blob/master/src/library/stats/R/t.test.R