Introduction to Computational BioStatistics with R: scripts & libraries

Erik Spence

SciNet HPC Consortium

24 September 2020

Today's slides

To find today's slides, go to the "Introduction to Computational BioStatistics with R" page, on the right, under Lectures, "Scripts".

`https://support.scinet.utoronto.ca/education`

Today's class

Today we will visit the following topics:

- Scripts.
- Libraries.
- Saving data.

Creating your own collections of functions

As you develop your research you will build a collection of functions which do all sorts of wonderful things. How should you manage your collections of functions?

- You should only have one copy of any given function. Put it in a file and grab it when you need it.
- The one copy you keep should be well-tested, well-commented, and trustworthy.
- If you have five copies of a function, how do you remember which one you should use?
- Keep your functions in files. Keep related functions in the same file.
- Name your functions and files sensibly.

But how do I access a function once I've written it?

Using your libraries of functions

```r
# MyUtilities.R

# Double and subtract 3.
doubleAndSubtract3 <- function(x) {
  return(2 * x - 3)
}

# ###############################

# Get the min, and add seven.
minPlusSeven <- function(x) {
  return(min(x) + 7)
}
```

```
> doubleAndSubtract3(10)
Error: could not find function "doubleAndSubtract3"
>
> source("MyUtilities.R")
>
> doubleAndSubtract3(10)
[1] 17
>
> minPlusSeven(8:12)
[1] 15
>
```

Use the "source" command.

Where am I?

Sometimes you need to know where you are, so that you can find data, and other things.

- getwd(): get the working directory.
- setwd('somedir'): set the working directory to "somedir".
- dir(): list the contents of the working directory.
- ls(): list the existing variables (you won't see built-in variables, such as pi, letters).

```
>
> getwd()
[1] "/c/Users/scinet"
>
> setwd('temp')
>
> getwd()
[1] "/c/Users/scinet/temp"
>
> dir()
test.R
>
> ls()
"doubleAndSubtract3"     "minOrZero"
>
```

Reloading modified functions

Suppose you're developing some code, editing a file in real time.

- Every time you resave the code you must 're-source' the file.
- R will continue to use the last version of the code which was sourced.

```
# sqrcode.R
# Version 1.

squared <- function(x) {
  return(x*2)
}
```

```
# sqrcode.R
# Version 2.

squared <- function(x) {
  return(x**2)
}
```

```
> source("sqrcode.R")
> squared(4)
[1] 8
>
> # fix the error
>
> source("sqrcode.R")
> squared(4)
[1] 16
```

Using non-standard R libraries

It is common to use packages that have been written by the R community.

- By default, even if you've installed a non-standard R package, non-standard R packages are not immediately available to use.
- To make them available, you must use the "library" function.
- This loads all the functions and variables of that package into the working environment.

```
>
> data(chiroptera)
Warning message:
In data(chiroptera) : data set chiroptera not found
>
> library(ape)
>
> data(chiroptera)
>
```

The 'data' function not only loads the default R data sets. It also can load specialty data sets which come with other packages.

Installing R packages

It's crazy-easy to install packages which you don't already have.

- To load a non-default library, use the 'library' function.
- To install non-standard packages, use 'install.packages'.
- Do not put "install.packages" commands in your scripts. Install needed packages by hand at the R prompt.

```
> library('fun')
Error in library("fun") : there is no package called fun
>
> install.packages("fun")
Installing package into '/home/ejspence/lib/R'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session
⋮
** testing if installed package can be loaded
* DONE (fun)
>
> library('fun')
>
```

Commenting code

Comment your code!

- Comments are notes which you put into your scripts which describe:
  - What you're doing.
  - Why you're doing it.
  - The assumptions that you're making.
  - The plan for the code.
- All languages have some means of commenting code.
- Even if it's obvious what you're doing, comment it anyway. You never know who might end up reading your code.
- If you need some help getting started with comments, drop a knock-knock joke into the code.
- Starting with the second assignment, we expect you to comment, and comment well, all the code in your homework assignments.
- In R (and Python, and bash), comments start with #.

Using R scripts

Once you have a series of commands which you will need to run repeatedly you should save them in a script.

- A script is just a list of commands that you want the R interpreter to execute.
- It's as if you are running the commands at the command line yourself.
- By saving your commands in a script, you'll remember what you did six months from now.
- Though 'A' and 'b' are global variables, they are being passed into the 'solve' command explicitly.
- Be careful if you copy-and-paste from the R prompt into an editor. Meaning, do not copy the ">" prompts into your script.

```
# myscript.R

# Create the matrix.
A <- matrix(rnorm(9),
            nrow = 3
            ncol = 3)

# Create b.
b <- 1:3

# Solve.
x <- solve(A, b)

cat("the answer is", x, "\n")
```

Running R scripts

Once you have a script, how do you run it? There are two options for running your script at the Linux command line:

- R CMD BATCH myscript.R. Note that by default this will generate a file called "myscript.Rout", which contains what would have been seen on the screen had you run the commands by hand.
- Rscript myscript.R. This is a better option, as it runs as a proper script.

```
[ejspence.mycomp]
[ejspence.mycomp] R CMD BATCH myscript.R
[ejspence.mycomp]
[ejspence.mycomp] Rscript myscript.R
The answer is -1.820291 -0.9132152 0.3703467
[ejspence.mycomp]
```

"Rscript" must be in your "PATH" for this to work.

Using R scripts, continued

A few notes about R scripts and writing your libraries of functions:

- By convention R script files have a ".R" file extension.
- Use comments, so you'll remember what you're trying to do! There are never too many comments. (Put them in your homework!)
- Document your functions, so it's easier to remember how to use them!
- R scripts are written in raw text, so be sure to use a text editor, not a word processor. Atom, Brackets, Sublime, Emacs, vi, nano are good. use
- When you use the "source" command, the whole file is read and executed as if you were typing the lines at the R prompt.
- You may be able to run your scripts within an IDE (like RStudio). It's better to learn to run them on a command line, so that you can run the scripts on Unix-based machines.

Utility files and driver programs

The general framework we will use for this course is to have "utilities files" and "driver scripts".

- The utilities files will contain the definitions of the functions you need to work on your data.
- It is common to have multiple utilities files, containing functions for performing different types of analyses.
- The driver script "drives" the analysis, invoking the functions in the utilities files as needed to accomplish whatever you are doing.
- The functions in the utilities file are often used by a variety of different driver scripts.

```r
# MyUtilities.R

# Double and subtract 3.
doubleAndSubtract3 <- function(x)
  {return(2 * x - 3)}
```

```r
# MyDriver.R

# Get the functions.
source("MyUtilities.R")

# Create the vector.
myvar <- 1:20

# Get the result.
b <- doubleAndSubtract3(myvar)

cat("the answer is", b, "\n")
```

Command line arguments

How do I access bash command line arguments from within an R script?

- use the "commandArgs" function.
- If you don't put "trailingOnly = T" you'll get the full Rscript command, along with many flags, along with the command line arguments you're after.
- Note that you can only test this from the bash prompt, not from within R or Rstudio.

```
# myscript2.R
args <- commandArgs(trailingOnly = TRUE)
cat("The type of args is", typeof(args), ".\n")
cat("The number of args is", length(args), ".\n")
cat("The command line arguments are", args, ".\n")
```

```
[ejspence.mycomp]
[ejspence.mycomp] Rscript myscript2.R pants 3.2
The type of args is character .
The number of args is 2 .
The command line arguments are pants 3.2 .
[ejspence.mycomp]
```

Command line arguments, continued

Be careful:

- The return value of "commandArgs" is a vector.
- If you want to compare the first argument to something else, you should directly reference the first argument.
- You will lose marks on your assignments if you don't reference specific argument indices in your scripts.

```
# myscript3.R
args <- commandArgs(trailingOnly = TRUE)

# Incorrect!
pants.check <- (args == "pants")

# Correct!
pants.check <- (args[1] == "pants")
```

```
[ejspence.mycomp] Rscript myscript3.R a
[ejspence.mycomp]
[ejspence.mycomp] Rscript myscript3.R a b
Warning message:
In if (args == "pants") {:
the condition has length > 1 and only the first element
will be used
[ejspence.mycomp]
```

Command line arguments, continued more

When comparing, make sure you are comparing things of the same type!

- The vector returned by "commandArgs" is a vector of STRINGS.
- If you want to compare a command line argument to something else, you should make sure your are comparing strings to strings.
- Alternatively, convert the command line argument string to another form if you want to compare to another form.

```
# myscript4.R
args <- commandArgs(trailingOnly = TRUE)

# Incorrect!
arg.check <- (args[1] == 1)

# Correct!
arg.check <- (args[1] == "1")

# Also correct, though dangerous!
arg.check <- (as.numeric(args[1]) == 1)
```

You will lose marks on your assignments if you compare two variables, such as the command line argument, of different types (string, numeric, boolean).

Using 'save' and 'load'

Printing results is fine, but at the end of the day you need to save your results to a file. Don't use a CSV file!

- You can save variables using the 'save' function.
- To load saved data, use 'load'.
- Note that your loaded data will overwrite any existing variables of the same name.

```
> a <- 10
> b <- 20
>
> save(a, b, file = "mydata.Rdata")
>

exit and come back

>
> load("mydata.Rdata")
>
> a
[1] 10
>
> b
[1] 20
>
```

Workspace management

You can also save the state of your R session:

- You are working in a "workspace". To save your workspace for next time, use save.image(). This will put your image in a file named ".Rdata".
- To load a previous workspace, use 'load'.
- Note that your loaded image will append to, and overwrite, you current workspace.
- R will ask if you want to save your workspace when you try to exit.

```
> # do a bunch of stuff
> save.image()
>
> # or alternatively
> save.image(file = 'myimage.Rdata')
>
> load("myimage.Rdata")
>
```

Enough to get started

There's obviously a lot more to learn about functions, scripting and libraries. Nonetheless, this is enough functionality to get you started, and to complete the second homework assignment.

Note that the second homework assignment has been posted.