Introduction to R

RSTR-seq edition

Kim Dill-McFarland, kadm@uw.edu

version October 28, 2021

Contents

Overview

In this workshop, we introduce you to R and RStudio at the beginner level. In it, we cover:

- R and RStudio including projects, scripts, and packages
- The help function
- Reading in data as a data frame and RData
- Data types

We will do all of our work in RStudio. RStudio is an integrated development and analysis environment for R that brings a number of conveniences over using R in a terminal or other editing environments.

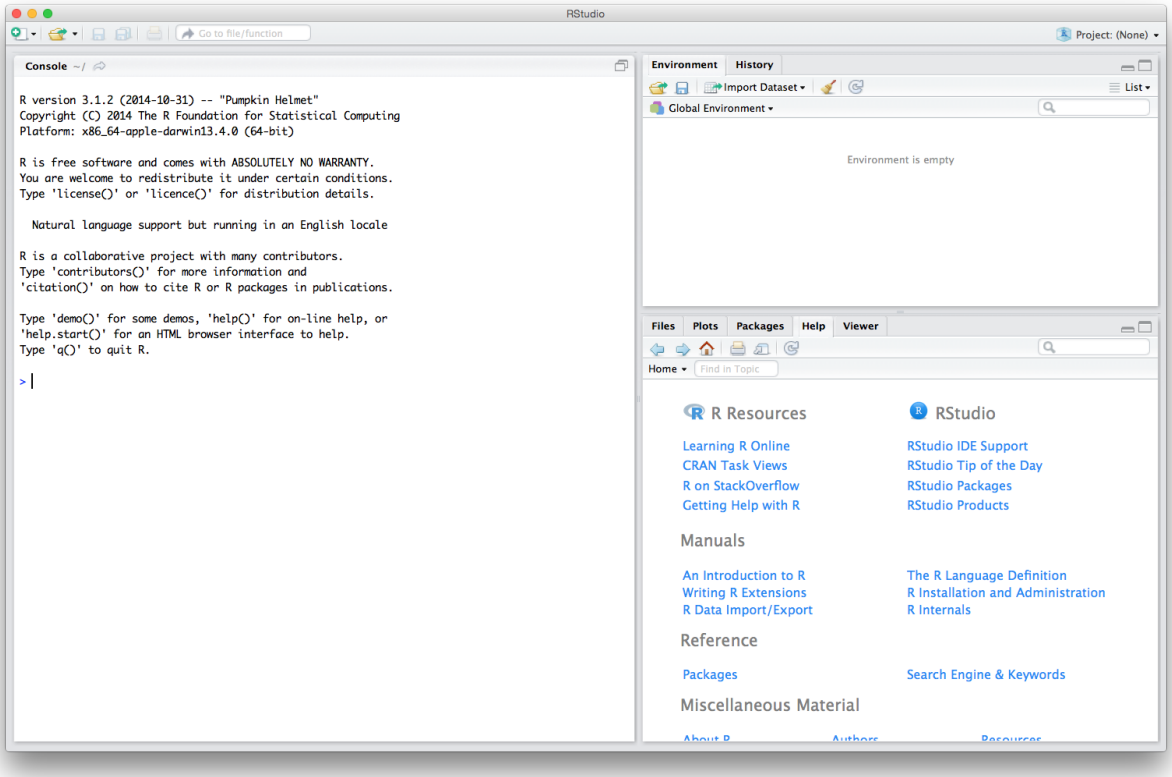
During the workshop, we will build an R script together, which will be posted as 'live_notes' after the workshop here.

Prior to the workshop

Please install R and RStudio. See the setup instructions for more details.

A Tour of RStudio

When you start RStudio, you will see something like the following window appear:



Notice that the window is divided into three "panes":

- Console (the entire left side): this is your view into the R engine. You can type in R commands here and see the output printed by R. (To make it easier to tell them apart, your input is printed in blue, while the output is black.) There are several editing conveniences available: use up and down arrow keys to go back to previously entered commands, which can then be edited and re-run; TAB for completing the name before the cursor; see more in online docs.
- Environment/History (tabbed in upper right): view current user-defined objects and previously-entered commands, respectively.
- Files/Plots/Packages/Help (tabbed in lower right): as their names suggest, these are used to view the contents of the current directory, graphics created by the user, install packages, and view the built-in help pages.

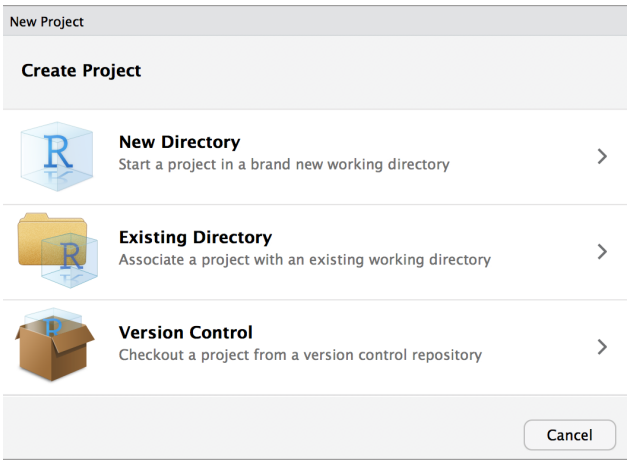
To change the look of RStudio, you can go to Tools -> Global Options -> Appearance and select colors, font size, etc. If you plan to be working for longer periods, we suggest choosing a dark background color scheme to save your computer battery and your eyes.

RStudio Projects

Projects are a great feature of RStudio. When you create a project, RStudio creates an `.Rproj` file that links all of your files and outputs to the project directory. When you import data, R automatically looks for the file in the project directory instead of you having to specify a full file path on your computer like `/Users/username/Desktop/`. R also automatically saves any output to the project directory. Finally, projects allow you to save your R environment in `.RData` so that when you close RStudio and then re-open it, you can start right where you left off without re-importing any data or re-calculating any intermediate steps.

RStudio has a simple interface to create and switch between projects, accessed from the button in the top-right corner of the RStudio window. (Labeled "Project: (None)", initially.)

Create a Project Let's create a project to work in for this workshop. Start by clicking the "Project" button in the upper right or going to the "File" menu. Select "New Project" and the following will appear.



You can either create a project in an existing directory or make a new directory on your computer - just be sure you know where it is.

After your project is created, navigate to its directory using your Finder/File explorer. You will see the `.RProj` file has been created.

To access this project in the future, simply double-click the `RProj` and RStudio will open the project or choose File > Open Project from within an already open RStudio window.

R Scripts

R script files are the primary way in which R facilitates reproducible research. They contain the code that loads your raw data, cleans it, performs the analyses, and creates and saves visualizations. R scripts maintain a record of everything that is done to the raw data to reach the final result. That way, it is very easy to write up and communicate your methods because you have a document listing the precise steps you used to conduct your analyses. This is one of R's primary advantages compared to traditional tools like Excel, where it may be unclear how to reproduce the results.

Generally, if you are testing an operation (*e.g.* what would my data look like if I applied a log-transformation to it?), you should do it in the console (left pane of RStudio). If you are committing a step to your analysis (*e.g.* I want to apply a log-transformation to my data and then conduct the rest of my analyses on the log-transformed data), you should add it to your R script so that it is saved for future use.

Additionally, you should annotate your R scripts with comments. In each line of code, any text preceded by the `#` symbol will not execute. Comments can be useful to remind yourself and to tell other readers what a specific chunk of code does.

Let's create an R script (File > New File > R Script) and save it as `live_notes.R` in your main project directory. If you again look to the project directory on your computer, you will see `live_notes.R` is now saved there.

We will work together to create and populate the `live_notes.R` script throughout this workshop.

R packages

CRAN R packages are units of shareable code, containing functions that facilitate and enhance analyses. Let's install `tidyverse`, which is actually a meta-package containing several packages useful in data manipulation and plotting. Packages are typically installed from CRAN (The Comprehensive R Archive Network), which is a database containing R itself as well as many R packages. Any package can be installed from CRAN using the `install.packages` function. You can input this into your console (as opposed to `live_notes.R`) since once a package is installed on your computer, you won't need to re-install it again.

```
install.packages("tidyverse", Ncpus=2)
```

This can take several minutes.

After installing a package, and *every time* you open a new RStudio session, the packages you want to use need to be loaded into the R workspace with the `library` function. This tells R to access the package's functions and prevents RStudio from lags that would occur if it automatically loaded every downloaded package every time you opened it.

```
# Data manipulation and visualization
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.5     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.0.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Bioconductor Bioconductor is another repository of R packages. It has different requirements for upload and houses many of the biology-relevant packages. To install from Bioconductor, you first install its installer from CRAN.

```
install.packages("BiocManager")
```

Then install your package of choice using it's installer. Here, we install `limma`, a package for analysis of microarray and RNA-seq data.

If prompted, say `a` to "Update all/some/none? [a/s/n]" and `no` to "Do you want to install from sources the packages which need compilation? (Yes/no/cancel)"

```
BiocManager::install("limma")
```

Getting started

Before doing anything in R, it is a good idea to set your random seed. Your analyses may not end up using a seed but by setting it, you ensure that *everything* is exactly reproducible.

```
set.seed(4389)
```

Organize data

Create a directory called `data` and move the 2 data files into this directory.

Loading data into R

Data frames from `.csv`, `.tsv`, etc.

One of R's most essential data structures is the data frame, which is simply a table of `m` columns by `n` rows. First, we will read in the RNA-seq metadata into RStudio using the base R `read.table` function.

Each R function follows the following basic syntax, where `Function` is the name of the function.

```
Function(argument1=..., argument2=..., ...)
```

`read.table` has many arguments; however, we only need to specify 3 arguments to correctly read in our data as a data frame. For our data, we will need to specify:

- `file` - gives the path to the file that we want to load from our working directory (current project directory).
- `sep` - tells R that our data are comma-separated
- `header` - tells R that the first row in our data contains the names of the variables (columns).

We will store the data as an *object* named `meta` using the assignment operator `<-`, so that we can re-use it in our analysis.

```
# read the data and save it as an object
meta <- read.table(file="data/RSTR_meta_subset.csv",
                   sep=",", header=TRUE)
```

Now whenever we want to use these data, we simply call `meta`

Help function

You can read up about the different arguments of a specific function by typing `?Function` or `help(Function)` in your R console.

```
?read.table
```

You will notice that there are multiple functions of the `read.table` help page. This include similar and related functions with additional options. For example, since our data are in `.csv` format, we could've instead read them into R with `read.csv` which assumes the options `sep=",", header=TRUE` by default.

```
# read the data with different function
meta <- read.csv(file="data/RSTR_meta_subset.csv")
```

Complex data from `.RData`

You may have data that do not fit nicely into a single table or into a table at all (like plots). You can save these as `.RData`, which can be loaded directly into R. You can also save multiple tables and/or other objects in a single `.RData` file to make loading your data quick and easy. Moreover, `.RData` are automatically compressed so they take up less storage space than multiple tables.

```
load("data/RSTR_data_clean_subset.RData")
```

Notice that these data appear already named in your R environment as `dat`. Object names are determined when saving so be sure to create short but descriptive names before saving to `.RData`.

See the objects data type with

```
class(dat)
```

```
## [1] "EList"
## attr(,"package")
## [1] "limma"
```

Data types

Simple

Let's return to the simpler metadata for now. This data frame consists of 20 rows (observations) and 6 columns (variables). You can see this quickly using the dimension function `dim`

```
dim(meta)
```

```
## [1] 20  6
```

Each column and each row of a data frame are individual R vectors. R vectors are one-dimensional arrays of data. For example, we can extract column vectors from data frames using the `$` operator.

```
# Extract patient IDs
meta$FULLIDNO
```

```
##  [1] "92527-1-02" "92527-1-08" "92527-1-08" "84437-1-02" "84437-1-02"
##  [6] "91587-1-04" "91587-1-04" "84457-1-02" "91360-1-04" "84457-1-02"
## [11] "91360-1-04" "84317-1-03" "84317-1-03" "89902-1-07" "89902-1-07"
## [16] "92527-1-02" "89448-1-04" "89448-1-04" "84427-1-02" "84427-1-02"
```

R objects have several different classes (types). Our data frame contains 2 R data types. The base R `class` function will tell you what data type an object is.

```
class(meta)
```

```
## [1] "data.frame"
```

```
class(meta$libID)
```

```
## [1] "character"
```

```
class(meta$lib.size)
```

```
## [1] "numeric"
```

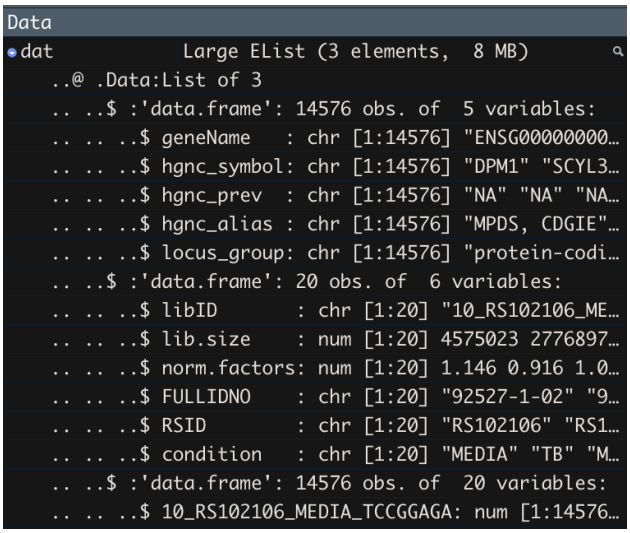
We see that our `libID` column is `character`, meaning it is non-numeric. On the other hand, `lib.size` is `numeric`, meaning a number.

Common data types not found in these data include the following. We will see these later on.

- `factor`: non-numeric value with a set number of unique levels
- `integer`: whole number numeric
- `logical`: TRUE/FALSE designation

Complex (S3, S4)

Now, let's look at the `limma EList` data. These data are in `S3` format meaning they have 3 dimensions. In essence, they are a list of multiple data frames and vectors. If you click on the `dat` object in your Environment tab, you will see multiple pieces.



All 3 pieces are data frames. You can again see this with `class` only this time you specify a part of the `dat` object with `$`

```
class(dat$genes)
```

```
## Loading required package: limma
```

```
## [1] "data.frame"
```

```
class(dat$E)
```

```
## [1] "data.frame"
```

```
class(dat$targets)
```

```
## [1] "data.frame"
```

Notice that you get the message `Loading required package: limma`. If you did not have `limma` installed, you could work with these data because they are a data type specific to `limma`.

Or going deeper, you can specify one column in the `genes` data frame

```
class(dat$genes$hgnc_symbol)
```

```
## [1] "character"
```

Of note, working with `S4` objects is very similar to `S3` except that they are accessed with `@` instead of `$`. However, we will not use `S4` in this workshop.

Working with vectors and data frames

Operating on vectors

A large proportion of R functions operate on vectors to perform quick computations over their values. Here are some examples:

```
# Compute the variance of library size
var(dat$targets$lib.size)
```

```
## [1] 1.905442e+13
```

```
# Find whether any samples have greater than 10 million sequences
dat$targets$lib.size > 10E6
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [13]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
# Find the unique values of metadata's condition
unique(meta$condition)
```

```
## [1] "MEDIA" "TB"
```

Using the correct class

Functions executed on an object in R may respond exclusively to one or more data types or may respond differently depending on the data type. When you use the incorrect data type, you will get an error or warning message. For example, you cannot take the mean of a factor or character.

```
# Compute the mean of libID
mean(meta$libID)
```

```
## Warning in mean.default(meta$libID): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

Subsetting vectors and data frames

Since vectors are 1D arrays of a defined length, their individual values can be retrieved using vector indices. R uses 1-based indexing, meaning the first value in an R vector corresponds to the index 1. (Importantly, if you use python, that language is 0-based, meaning the first value is index 0.) Each subsequent element increases the index by 1. For example, we can extract the value of the 5th element of the `libID` vector using the square bracket operator `[ ]` like so.

```
meta$libID[5]
```

```
## [1] "RS102306_MEDIA"
```

In contrast, data frames are 2D arrays so indexing is done across both dimensions as `[rows, columns]`. So, we can extract the same oxygen value directly from the data frame knowing it is in the 5th row and 1st column.

```
meta[5, 1]
```

```
## [1] "RS102306_MEDIA"
```

The square bracket operator is often used with logical vectors (TRUE/FALSE) to subset data. For example, we can subset our metadata to all `MEDIA` observations (rows).

```
# Create logical vector for which lib.size values are > 10 million
logical.vector <- meta$condition == "MEDIA"
#View vector
logical.vector
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
## [13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
#Apply vector to data frame to select only observations where the logical vector is TRUE
meta[logical.vector, ]
```

```
##             libID lib.size norm.factors   FULLIDNO     RSID condition
## 1  RS102106_MEDIA  4575023     1.146246 92527-1-02 RS102106     MEDIA
## 3  RS102531_MEDIA  4258343     1.073279 92527-1-08 RS102531     MEDIA
## 5  RS102306_MEDIA 12415741     1.023636 84437-1-02 RS102306     MEDIA
## 7  RS102087_MEDIA  4642072     1.123349 91587-1-04 RS102087     MEDIA
## 10 RS102244_MEDIA 10744984     1.021158 84457-1-02 RS102244     MEDIA
## 11 RS102521_MEDIA 14509963     1.034990 91360-1-04 RS102521     MEDIA
## 13 RS102340_MEDIA 10601432     1.082688 84317-1-03 RS102340     MEDIA
## 15 RS102548_MEDIA 17242699     1.332383 89902-1-07 RS102548     MEDIA
## 17 RS102469_MEDIA 13666571     1.090187 89448-1-04 RS102469     MEDIA
## 19 RS102484_MEDIA 15120282     1.090477 84427-1-02 RS102484     MEDIA
```

Subsetting is extremely useful when working with large data. We will learn more complex subsets on day 2 using the tidyverse. But first...

Quick reference: Conditional statements

Statement	Meaning
`<-`	Assign to object in environment
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`%in%`	In or within
`is.na()`	Is missing, *e.g* NA
`!is.na()`	Is not missing
`&`	And
`|`	Or

Exercises: Part 1

1. Using help to identify the necessary arguments for the log function, compute the natural logarithm of 4, base 2 logarithm of 4, and base 4 logarithm of 4.

Using the `meta` data frame:

2. Using an R function, determine what data type the `norm.factors` variable is.
3. Using indexing and the square bracket operator `[]`:
   - determine what `RSID` value occurs in the 20th row
   - return the cell where `lib.size` equals 14509963
4. Subset the data to observations where `RSID` equals "RS102521" or "RS102484". *Hint*: Use a logical vector.

Additional resources

- R cheatsheets also available in RStudio under Help > Cheatsheets
- Rladies Seattle Not just for ladies! A pro-actively inclusive R community with both in-person and online workshops, hangouts, etc.

- The Carpentries
- TidyTuesday A weekly plotting challenge
- R code club Dr. Pat Schloss opened his lab's coding club to remote participation.
- Seattle useR Group

R session

```
sessionInfo()
```

```
## R version 4.1.1 (2021-08-10)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Big Sur 10.16
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] limma_3.48.3    forcats_0.5.1   stringr_1.4.0   dplyr_1.0.7
##  [5] purrr_0.3.4     readr_2.0.2     tidyr_1.1.4     tibble_3.1.5
##  [9] ggplot2_3.3.5   tidyverse_1.3.1
##
## loaded via a namespace (and not attached):
##  [1] tidyselect_1.1.1 xfun_0.27        haven_2.4.3      colorspace_2.0-2
##  [5] vctrs_0.3.8      generics_0.1.0   htmltools_0.5.2  yaml_2.2.1
##  [9] utf8_1.2.2       rlang_0.4.12     pillar_1.6.4     glue_1.4.2
## [13] withr_2.4.2      DBI_1.1.1        dbplyr_2.1.1     modelr_0.1.8
## [17] readxl_1.3.1     lifecycle_1.0.1  munsell_0.5.0    gtable_0.3.0
## [21] cellranger_1.1.0 rvest_1.0.2      evaluate_0.14    knitr_1.36
## [25] tzdb_0.1.2       fastmap_1.1.0    fansi_0.5.0      broom_0.7.9
## [29] Rcpp_1.0.7       scales_1.1.1     backports_1.2.1  jsonlite_1.7.2
## [33] fs_1.5.0         hms_1.1.1        digest_0.6.28    stringi_1.7.5
## [37] grid_4.1.1       cli_3.0.1        tools_4.1.1      magrittr_2.0.1
## [41] crayon_1.4.1     pkgconfig_2.0.3  ellipsis_0.3.2   xml2_1.3.2
## [45] reprex_2.0.1     lubridate_1.8.0  rstudioapi_0.13  assertthat_0.2.1
## [49] rmarkdown_2.11   httr_1.4.2       R6_2.5.1         compiler_4.1.1
```
