Day 3, Session 1: R and RStudio basics

Jessica Williams-Nguyen and Brian D. Williamson

EPI/BIOST Bootcamp 2018

25 September 2018

Learning objectives

By the end of this session, you should be able to

- **organize** your files for a data analysis
- **download** data from the internet
- **load** data into an R workspace
- **explore** your data (subset, index, plot, summarize)
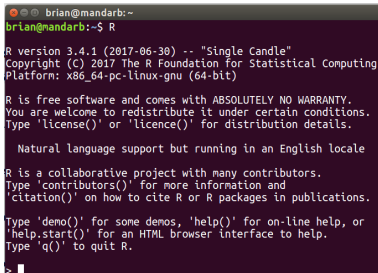- **load** R packages

The interface

We interact with both R and RStudio using an interface. For R, this is the command line. For RStudio, this is the graphical user interface.

Both of these concepts deserve a bit of introduction, since familiarity with the interface makes using R and RStudio much easier!

Command line R

The most basic form of R is from the command line (Terminal in Mac/Linux, Command Prompt in Windows).

After installing R, typing R in the command line will open an R session:

The text gives you the version number (here, 3.4.1) and some other (not necessary for our purposes) information.

Command line R

You also get a prompt, the >, where you can enter commands.

When you first open R, you have a limited number of functions available for use. These include things like `mean()`, `median()`, and `read.table()`.

The command line version of R is most useful (in my experience) for computing-intensive tasks. Data analyses are best done in RStudio.

RStudio

When you first open RStudio, you are met with a blank set of four panes:

RStudio
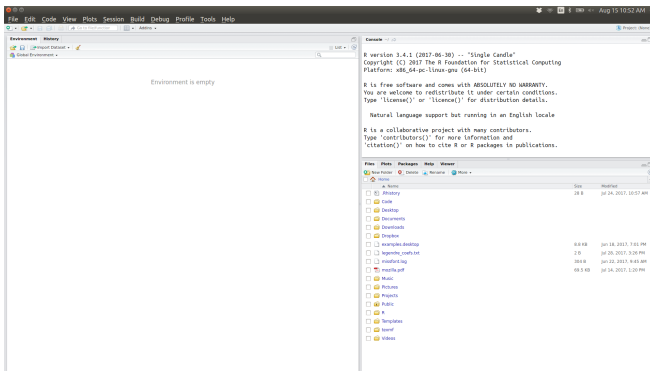
You can edit pane layout under `Tools > Global Options > Pane Layout`. My preference is to have the source in the upper left, console in the upper right, environment in the lower left, and viewer in the lower right.

You'll notice that on the previous slide, there were only three panes visible: since we haven't opened or created any files to edit, the source pane is not currently open.

RStudio: the console

Is it a coincidence that the screenshot has exactly the same text as we saw on the command line? No!

The console performs identically to the command line. Any commands that we wish to enter go through the console, and any results that are output by these commands will appear in the console.

Example: the console

Typing 47 at the > symbol (from now on, called the execution line) and hitting Enter on the keyboard yields the following:

```
> 47
[1] 47
```

Since R is a functional programming language, typing 47 and hitting Enter is the same thing as using the print() function:

```
> print(47)
[1] 47
```

The output is displayed as a vector, one of the fundamental data structures in R. The [1] helps to tell which element of the vector we are looking at (which is most useful if the vector spans multiple lines in the console).

Example: heights and weights

The console is the workhorse of RStudio. If we wanted, we could work exclusively here, but then reproducibility is difficult...

Let's say we have data on the height and weight of five (imaginary) individuals. A simple way of reading the data into R is by creating two objects, `ht` and `wt`, and assigning the numerical values that we have collected for each individual; in the console, type

1. `ht <- c(72,65,84,73,68)`, followed by hitting the Enter key, (creates the `ht` object)
2. `wt <- c(165,120,210,180,125)`, followed by hitting the Enter key (creates the `wt` object)

Each of these commands assigns a value to an object, using the special `<-` command. Values go on the right-hand side, and objects go on the left.

Example: heights and weights

Both `ht` and `wt` are also vectors! We create vectors using the `c()` function, which concatenates scalars (single numbers, or single strings like `"Hello"`) into vectors.

In words, `ht <- c(72,65,84,73,68)` means: concatenate the numbers 72, 65, 84, 73, and 68 into a vector, and assign that vector to a variable (or object) called `ht`.

Typing these commands into the console allows us to manipulate both `ht` and `wt`; in particular, they can be used as arguments into many R functions.

Example: heights and weights

However, these objects are part of the current R environment: this means that we can only use them while the current session of R is open. If we exit RStudio, we will have to re-enter the two lines defining both `ht` and `wt`.

For more complicated analyses, this can lead to a lot of confusion. Also, just typing in the console can lead us to forget what steps we took to get a certain figure or table.

Organizing commands into scripts allows us to save exactly what we did during a given data analysis, which makes reproducible research easy. This helps both your future self and your collaborators, so that you can share what you did!

RStudio: the text editor ("Source")

The text editor (which RStudio calls the source) is the second most powerful part of the IDE. This allows us to edit and save files, and to run commands directly from the text file into the console.

To create a new R script, either type Ctrl+Shift+N or go to File > New > R script. There are many other options here (most notably R markdown, which we won't cover but is incredibly useful). This brings up the source pane.

At the top of the source pane are a set of buttons (with associated keyboard shortcuts) that will save the current file, run sections of code, or run the entire document.

R scripts

R scripts are collections of R commands and comments. These are run in the console to manipulate objects and produce output.

To create a comment, type a # symbol. Anything on a line following a # will not be run by R.

I like to include a preamble in all of my R scripts, which gives me information on when I created the file, what it does, whether or not it takes any input or produces output, and what I have changed since I started. Here is a sample:

```
################################################################################################
##
## FILE: <insert file name here>.R
##
## CREATED: <insert date here> by <your name here>
##
## PURPOSE: <Give a brief description of what the code in this file is supposed to do>
##
## INPUTS: <What inputs? Where does the data come from?>
##
## OUTPUTS: <What outputs? Plot/table names and locations>
##
## UPDATES:
## DDMMYY INIT COMMENTS
## ------ ---- --------
## <date> <inits> <What did you change?>                                                       14
################################################################################################
```

Running commands from R scripts

There are many options for running commands using your R script, either in individual lines or in blocks.

You can run individual lines by placing your cursor on the line and

- hitting the green `Run` button, or

- hitting `Ctrl+Enter` (Windows/Linux) or `Cmd+Enter` (Mac)

Similarly, you run multiple lines by highlighting all desired lines and doing one of the two options described above.

Example: heights and weights

We can now enter

```
ht <- c(72,65,84,73,68)
wt <- c(165,120,210,180,125)
```

into a new R script. Saving this as `ht_wt_analysis.R` is a start to analyzing these data! The script now looks like:

```
##########################################################################################
##
## FILE: ht_wt_analysis.R
##
## CREATED: 15 August 2017 by Brian Williamson
##
## PURPOSE: Teach some R basics using height and weight data on 5 imaginary individuals
##
## INPUTS: None
##
## OUTPUTS: None
##
## UPDATES:
## DDMMYY INIT COMMENTS
## ------ ---- --------
##########################################################################################

ht <- c(72,65,84,73,68)
wt <- c(165,120,210,180,125)
```

RStudio: environment, history, etc.

The basic environment pane allows us to do three things:

- view which objects (data sets, etc.) exist in the `Global Environment`

- view the history of which commands were entered in the console

- import datasets (using the buttons on the top of the pane)

The `Global Environment` is where commands that are executed in the console typically live — intermediate objects created within functions (but not returned) live within function-specific environments.

The full command history for a given session is accessed in the `History` tab; this can be cycled through in the console using the up/down arrow keys.

Example: heights and weights

Execute the commands in `ht_wt_analysis.R` using one of the two options we described earlier (`Run` button or `Ctrl/Cmd+Enter`).

The environment pane then shows that `ht` and `wt` are `Values` (not data sets since they are vectors), and the history pane shows that we have input these two lines into the console.
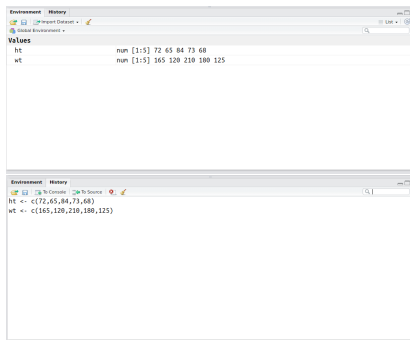
RStudio: files, plot viewer, packages, help

The final pane shows us a variety of files/plots:

- files in the current working directory

- plots generated from executed commands

- loaded packages

- help files that we access

We will cover packages and help files later on.

The working directory

Folders on your computer are also known as directories. These can contain data, documents, and subdirectories, among many other objects.

Your computer has a home directory, where all of your files (as the user) are stored, in different directories (e.g., `Documents`, `Pictures`, etc.). When you first log into your computer, you are in this directory.

The current working directory is the directory where you are currently working. To be a bit less obtuse, consider the following example:

You have a file called `ht_wt_analysis.R`, located in the `Epi-Biost-Workshop` directory, which is a subdirectory of `UW`, which is a subdirectory of `Documents`, which is part of your home directory. If you double-click `ht_wt_analysis.R`, which opens RStudio to edit the document, your current working directory is `<your home>/Documents/UW/Epi-Biost-Workshop`. 20

Changing the working directory using RStudio

Generally, when you open RStudio, your current working directory will be your home directory. This can make loading data that you have saved on your computer somewhat difficult, as we will discuss later.

One way to remedy this is to manually set your working directory using R/RStudio: either

- Using buttons:
  1. In the file viewer, navigate to the folder where your dataset resides
  2. Click on the More button (with the settings-type wheel next to it) under the file viewer tab
  3. Click Set as working directory
- Using code:
  1. Find the path to the folder where your data resides (e.g., /home/brian/Documents/UW/Epi-Biost-Workshop)
  2. Place the code setwd("<path to your data>") in your R script, or enter it at the command line

Introduction to R programming

We have already seen some examples of R programming: in the heights and weights example, we learned about the special `<-` command, which assigns a value to an object; and we learned about the `c()` command, which creates a vector.

These commands relate functions to objects — both of these are fundamental concepts in R programming.

Functions

R functions take in arguments and return values. A function is accessed by typing its name, followed by an open and closed set of parentheses: e.g., `quantile()`, a function to compute desired sample quantiles.

Arguments to functions go between the parentheses, and are separated by columns. You specify which object corresponds to which argument using =, e.g. `quantile(x = ht, probs = 0.5)` returns the median value of the `ht` object.

Values are returned by functions, and are generally either a combination of objects, plots, and printout. The value of the result of entering `quantile(x = ht, probs = 0.5)` into the console is a vector containing the number 72.

Objects

R objects are (basically) the result of calling functions. However, there are two general ways this can be done:

- loading data (e.g., from a text or .csv file)

- manipulating another object using a function

Both `ht` and `wt` are objects; we created them by manipulating raw numbers using the `c()` function.

Loading data

Data can be loaded using menus, buttons, or a function. The menus and buttons ultimately call a function; each data storage type (e.g., `.txt`, `.csv`, `.sas7bdat`, `.dta`) has its own function (and some functions come from special packages).

Regardless of how you load the data, once done it will show up in your environment. It can then be manipulated to display summary statistics, make plots, or do more complex analyses.

Loading data: menus

1. Go to `File > Import Dataset`
2. Select the type of data from the menu (e.g., `From CSV`)
3. Browse to the directory or website where the data are stored
4. Specify any input options
5. Edit the `Code Preview` section to rename the object, if necessary
6. Click `Import`

Loading data: buttons

1. In the `Environment` pane, click `Import Dataset`
2. Select the type of data from the menu (e.g., `From CSV`)
3. Browse to the directory or website where the data are stored
4. Specify any input options
5. Edit the `Code Preview` section to rename the object, if necessary
6. Click `Import`

Loading data: menus and buttons

Regardless of whether you use a menu or button to import data, there was a step where you had to look at the `Code Preview` box. When you click the `Import` button, this code is executed in the console, and the dataset shows up in your environment.

This means that, under the hood, both methods use R code to import data!

Also, both of these methods are hard to reproduce: each time you want to analyze a dataset (which may be multiple times on a problem set or during the course of a project/paper), you have to click through these buttons.

To make your code fully reproducible, embed the code to load the data directly in your R script! While this involves finding the correct functions, at the end of the day you have more control over how your data are handled.

Loading data: functions

1. Find the path to your data file
   - e.g.,
     `"/home/brian/Documents/UW/Epi-Biost-Workshop/ht_wt_data.txt"`
2. Based on the file extension (e.g., `.txt`, `.csv`, `.dta`), choose the appropriate function
3. Put the function call with the file path in your R script with the desired options
   - e.g., `read.table("/home/brian/Documents/UW/Epi-Biost-Workshop/ht_wt_data.txt", sep = "", header = TRUE)`
4. Run this line to load the data

Loading data: common functions and packages

File extension	Package	Function
.txt	utils (always loaded)	read.table()
		read.delim()
	readr	read_table()
		read_delim()
.csv	utils	read.csv()
	readr	read_csv()
.dta	haven	read_stata()
.sas7bdat	haven	read_sas()
.xlsx	readxl	read_excel()
.Rdata	base (always loaded)	load()
.Rds	base	readRDS()

Saving data

Once you have manipulated data to your satisfaction, you can also save data — often, this is a helpful part of the data analysis pipeline from raw data, through pre-processing, to the final analysis data.

To save data, you specify the path to the new file you wish to create, and use this as the first argument in the saving function. The appropriate save function simply replaces `read` with `save` in the functions on the previous page.

Example: heights and weights

We can load the height and weight data using the following code (replace <path to here> with the path to your folder):

```
dat <- read.table("<path to here>/Epi-Biost-Workshop/
                   day_3_session_1/data/ht_wt_data.txt",
                   header = TRUE)
```

If we look at the data (typing `dat` into the console and hitting Enter), we see that this is the same as the two vectors, `ht` and `wt`, combined into a dataset! Each row is an imaginary person, and the columns contain their height and weight.

Add this code to your R script, beneath the code assigning values to `ht` and `wt`.

Manipulating data

A large component of any data analysis is getting the data into a useable format for our analysis: this usually involves a combination of creating new variables and taking subsets of the data.

The two most common types of data you will see are vectors and data frames. We have already been introduced to vectors in the heights and weights example; data frames are generally datasets, where the columns are variables (each of these is a vector), and the rows are observations.

Thus by manipulating the columns of a data frame we create new variables or change existing ones; and by manipulating the rows of our data we restrict our attention to different subsets of the observations.

Manipulating data: accessing values in vectors

The easiest way to access data is from a vector: each vector is its own variable, and hence we just need to extract the correct element(s).

To access the ith element in the vector vec, we use square brackets; vec[i] accomplishes this. For example, ht[1] returns the first element in the ht vector, which is the height of our first imaginary individual.

To access a continuous range of elements i through l, use square brackets with a colon; vec[i:l] accomplishes this. For example, ht[1:4] returns the first four elements of ht.

To access a set of elements (i, k, n), use square brackets with c(); vec[c(i, k, n)] accomplishes this. For example, ht[c(1, 3, 5)] returns the first, third, and fifth elements of ht.

Manipulating data: accessing values in data frames

Data frames have both rows and columns. The dimensions of the data frame are first rows, then columns. We can access a single value in a data frame by first specifying the desired row, then the desired comma, within square brackets; for a data frame `dat` with 5 rows and 2 columns, we can get the first observation from the second column using `dat[1, 2]`.

For example, in the heights and weights data, `dat[1, 2]` returns the weight of the first individual.

We can use the same rules as those for vectors to select groups of rows and/or columns.

Manipulating data: accessing variables in data frames

There are three ways to access a variable in a data frame. For a data frame `dat`, with columns `x` and `y`, we can use:

- `$`; `dat$y` returns the variable `y` from the data frame `dat`

- the column number; `dat[, 2]` returns the variable `y` from `dat`

- the column name; `dat[, "y"]` returns the variable `y` from `dat`

The methods using the square brackets introduce a new concept: we are selecting all rows from `dat` (since there is nothing between the first bracket and the comma), and are selecting certain columns from `dat`.

Manipulating data: subsetting data frames

Often, we don't know exactly which rows we want to select. Using logical operations helps us select participants with certain characteristics. Logical operations return TRUE or FALSE.

The common logical operations are:

Operation	Meaning	Result
|	or	TRUE if any argument is TRUE
&	and	TRUE only if all arguments are TRUE
!	negation	FALSE if argument returns TRUE, TRUE else
==	equals	TRUE if the two arguments are equal

We can also make comparisons, using $<$, $>$, $<=$, $>=$.

Example: heights and weights

The following code loads the data as a data frame, and then: prints out the height column, and displays the data from participants whose weight is below 160 lbs (replace <path to here> with your own path):

```
## load using read.table
dat <- read.table("<path to here>/Epi-Biost-Workshop/
                   day_3_session_1/data/ht_wt_data.txt", header = TRUE)
## print out the height column multiple ways
dat$height
dat[, "height"]
dat[, 2]

## print out the observations with weight under 160
dat[dat$weight < 160, ]
```

R packages

Functions in R are available through packages. Each package contains a specific set of functions, and must be both installed and loaded before the function can be used.

Each package only has to be installed once, unless you upgrade your version of R. Packages are installed using the `install.packages()` function.

Each time you open a new R session, you have to re-load all R packages you wish to use in the session. Packages are loaded using the `library()` function.

However, some packages are always loaded each time you open R. These are `stats`, `graphics`, `grDevices`, `utils`, `datasets`, `methods`, and `base`.

Finding new R packages

In general, R packages are stored on one (or more) of three repositories: CRAN, under the `Packages` side-menu; Bioconductor, under the `Explore packages` menu; and individual programmer's GitHub pages (e.g., the repository for the devtools package, for easy R package creation).

You can install packages from CRAN using `install.packages()`; from Bioconductor by first running `source("https://bioconductor.org/biocLite.R")` and then using `biocLite()`; and from GitHub using the function `install_github()` from the `devtools` package.

Example: heights and weights

The following code reads in the height and weight data using the read_table() function from readr:

```
## install if necessary
# install.packages("readr")
## load the package; do this each time you re-open R!
library("readr")
## read in the data
dat <- read_table("<path to here>/Epi-Biost-Workshop/
                   day_3_session_1/ht_wt_data.txt", header = TRUE)
```