

2.7: R and Data

How to enter the data from within R

We now need to know how to enter data into R. Basic command is `c()` (shortcut of the word *concatenate*):

However, in that way your numbers will be forgotten because R does not remember anything which is not saved into *object*:

(Here we *created* an object `aa`, *assigned* to it vector of numbers from one to five, and then *printed* object with typing its name.)

If you want to create and print object simultaneously, use external parentheses:

(By the way, here we created `aa` object *again*, and R *silently* re-wrote it. R never gives a warning if object already exists!)

In addition to `c()`, we can use commands `rep()`, `seq()`, and also the colon `:` operator:

How to name your objects

R has no strict rules on the naming your objects, but it is better to follow some guidelines:

1. Keep in mind that R is *case-sensitive*, and, for example, `X` and `x` are different names.
2. For objects, use only English letters, numbers, dot and (possibly) underscore. Do not put numbers and dots in the beginning of the name. One of recommended approaches is double-letter (or triple-letter) when you name objects like `aa`, `jjj`, `xx` and so on.
3. In data frames, we recommend to name your columns (characters) with *uppercase letters and dots*. The examples are throughout of this book.
4. Do not reassign names already given to popular functions (like `c()`), reserved words (especially `T`, `F`, `NA`, `NaN`, `Inf` and `NULL`) and predefined objects like `pi`^[1], `LETTERS` and `letters`. If you accidentally did it, there is `conflict()` function to help in these situations. To see all reserved words, type `?Reserved`.

How to load the text data

In essence, data which need to be processed could be of two kinds: *text* and *binary*. To avoid unnecessary details, we will accept here that *text data* is something which you can read and edit in the *simple text editor* like Geany^[2]. But if you want to edit the *binary data*, you typically need a program which outputted this file in the past. Without the specific software, the binary data is not easy to read.

Text data for the statistical processing is usually text tables where every row corresponds with the table row, and columns are separated with *delimiters*, either invisible, like spaces or tab symbols, or visible, like commas or semicolons. If you want R to “ingest” this kind of data, is is necessary to make sure first that the data file is located within the same directory which R regards as a *working directory*:

If this is not the directory you want, you can change it with the command:

Note how R works with backslashes under Windows. Instead of one backslash, you need to enter *two*. Only in that case R under Windows will understand it. It is also possible to use slashes under Windows, similar to Linux and macOS:

Please always start each of your R session from changing working directory. Actually, it is not absolutely necessary to remember long paths. You can copy it from your file manager into R. Then, graphical R under Windows and macOS have rudimentary menu system, and it is sometimes easier to *change working directory though the menu*. Finally, collection `asmisc.r` contains function `Files()` which is the textual *file browser*, so it is possible to run `setwd(Files())` and then follow screen instructions^[3].

The next step after you got sure that the working directory is correct, is to check if your data file is in place, with `dir()` command:

It is really handy to separate data from all other stuff. Therefore, we assumed above that you have subdirectory `data` in you working directory, and your data files (including `mydata.txt`) are in that subdirectory. Please create it (and of course, create the working directory) if you do not have it yet. You can create these with your file manager, or even with R itself:

Now you can load your data with `read.table()` command. But wait a minute! You need to *understand the structure* of your file first.

Command `read.table()` is sophisticated but it is not smart enough to determine the data structure on the fly^[4]. This is why you need to check data. You can open it in any available simple text editor, in your Web browser, or even from inside R with `file.show()` or `url.show()` command. It outputs the data “as is”. This is what you will see:

(By the way, if you type `file.show("data/my` and press `Tab`, *completion* will show you if your file is here—if it is really here. This will save both typing file name and checking the presence with `dir()`.)

How did the file `mydata.txt` appear in your data subdirectory? We assume that you already downloaded it from the repository mentioned in the foreword. If you did not do it, please do it now. It is possible to perform with any browser and even with R:

(Within parentheses, left part is for URL whereas right tells R how to place and name the downloaded file.)

Alternatively, you can check your file directly from the URL with `url.show()` and then use `read.table()` from the same URL.

Now time finally came to load data into R. We know that all columns have names, and therefore use `head=TRUE`, and also know that the delimiter is the semicolon, this is why we use `sep=";"`:

Immediately after we loaded the data, we must check the new object. There are three ways:

Third way is to simply type `mydata` but this is not optimal since when data is large, your computer screen will be messed with content. Commands `head()` and `str()` are much more efficient.

To summarize, local data file should be loaded into R in *three steps*:

1. Make sure that you data is in place, with `dir()` command, `Tab` completion or through Web browser;
2. Take a look on data with `file.show()` or `url.show()` command and determine its structure;
3. Load it with `read.table()` command *using appropriate options* (see below).

How to load data from Internet

Loading remote data takes same three steps from above. However, as the data is not on disk but somewhere else, to check its presence, the best way is to *open it in the Internet browser* using URL which should be given to you; this also makes the second step because you will see its structure in the browser window. It is also possible to check the structure with the command:

Then you can run `read.table()` but with URL instead of the file name:

(Here and below we will sometimes skip creation of new object step. However, remember that you *must create new object* if you want to use the data in R later. Otherwise, the content will be shown and immediately forgotten.)

How to use read.table()

Sometimes, you want R to “ingest” not only column names but also row names:

(File `mydata1.txt`^[5] is constructed in the unusual way: its first row has three items whereas all other rows each have four items delimited with the *tab symbol*—“big invisible space”. Please do not forget to check that beforehand, for example using `file.show()` or `url.show()` command.)

Sometimes, there are both spaces (inside cells) and tabs (between cells):

If we run `read.table()` without `sep="t"` option (which is “separator is a tab”), R will give an error. Try it. But why did it work for `mydata1.txt`? This is because the *default* separator is *both* space and/or tab. If one of them used as the part of data, the other must be stated as separator explicitly.

Note also that since row names contain quote, quoting must be disabled, otherwise data will silently read in a wrong way.

How to know what separator is here, tab or space? This is usually simple as most editors, browsers and `file.show()` / `url.show()` commands visualize tab as a space which is much broader than single letter. However, do not forget to *use monospaced font* in your software, other fonts might be deceiving.

Sometimes, numbers have comma as a decimal separator (this is another worldwide standard). To input this kind of data, use `dec` option:

(Please note the shortcuts. Shortcuts save typing but could be dangerous if they match several possible names. There are only one `read.table()` argument which starts with `se`, but several of them start with `s` (e.g., `skip`); therefore it is impossible to reduce `se` further, into `s`. Note also that `TRUE` and `FALSE` are possible to shrink into `T` and `F`, respectively (but this is the only possible way); we will avoid this in the book though.)

When `read.table()` sees character columns, it converts them into *factors* (see below). To avoid this behavior, use `as.is=TRUE` option.

Command `scan()` is similar to `read.table()` but reads all data into only one “column” (one vector). It has, however, one unique feature:

(What did happen here? First, we entered `scan()` with *empty first argument*, and R changed its prompt to numbers allowing to type numerical data in, element after element. To finish, enter `empty row` (^{[6]}). One can paste here even numbers from the clipboard!)

How to load binary data

Functions from the `foreign` package (it is installed by default) can read data in MiniTab, S, SAS, SPSS, Stata, Systat, and FoxPro DBF binary formats. To find out more, you may want to call it first with command `library(foreign)` and then call help about all its commands `help(package=foreign)`.

R can upload images. There are multiple packages for this, one of the most developed is `pixmap`. R can also upload GIS maps of different formats including ArcInfo (packages `maps`, `maptools` and others).

R has its own *binary format*. It is very fast to write and to load^[7] (useful for big data) but impossible to use with any program other than R:

(Here we used several new commands. To save and to load binary files, one needs `save()` and `load()` commands, respectively; to remove the object, there is `rm()` command. To show you that the object was deleted, we used `exists()` command.)

Note also that everything which is written after “#” symbol on the same text string is a *comment*. R skips all comments without reading.

There are many interfaces which connect R to databases including MySQL, PostgreSQL and sqlite (it is possible to call the last one directly from inside R see the documentation for [RSQLite](#) and [sqldf](#) packages).

But what most users actually need is to load the *spreadsheet data* made with MS Excel or similar programs (like Gnumeric or LibreOffice Calc). There are three ways.

First way we recommend to all users of this book: convert Excel file into the text, and then proceed with `read.table()` command explained above^[8]. On macOS, the best way is likely to save data from spreadsheet as tab-delimited text file. On Windows and Linux, if you copy any piece of spreadsheet into clipboard and then paste it into text editor (including R script editor), it becomes the tab-delimited text. The same is possible in macOS but you will need to use some terminal editor (like nano).

Another way is to use external packages which convert binary spreadsheets “on the fly”. One is [readxl](#) package with main command `read_excel()`, another is [xlsx](#) package with main command `read.xlsx()`. Please note that these packages are not available by default so you need to *download and install* them (see below for the explanations).

How to load data from clipboard

Third way is to use clipboard. It is easy enough: on Linux or Windows you will need to *select* data in the open spreadsheet, *copy* it to clipboard, and then in R window *type* command like:

On macOS, this is slightly different:

(Ignore warnings about “incomplete lines” or “closed connection”. Package [clipr](#) unifies the work with clipboard on main OSes.)

“Clipboard way” is especially good when your data come out of non-typical software. Note also that entering `scan()` and then pasting from clipboard (see above) work the same way on all systems.

Summarizing the above, recommended data workflow in R might look like:

1. Enter data into the spreadsheet;
2. Save it as a text file with known delimiters (tab and semicolon are preferable), headers and row names (if needed);
3. Load it into R with `read.table()`;
4. If you must change the data in R, write it afterwards to the external file using `write.table()` command (see below);
5. Open it in the spreadsheet program again and proceed to the next round.

One of its big pluses of this workflow is the *separation between data editing and data processing*.

How to edit data in R

If there is a need to change existing objects, you could *edit* them through R. We do not recommend this though, spreadsheets and text editors are much more advanced than R internal tools.

Nevertheless, there is a *spreadsheet* sub-program embedded into R which is set to edit table-like objects (matrices or data frames). To start it on `bb` matrix (see above), enter command `fix(bb)` and edit “in place”. Everything which you enter will immediately change your object. This is somewhat contradictory with R principles so there is the similar function `edit()` which does not change the object but *outputs* the result to the R window.

For other types of objects (not table-like), commands `fix()` / `edit()` call internal (on Windows or macOS) or external (on Linux) text editor. To use external editor, you might need to supply an additional argument, `edit(..., editor="name")` where `name` could be any text editor which is available in the system.

R on Linux has vi editor as a default but it is too advanced for the beginner^[9]; we recommend to use nano instead^[10]. Also, there is a `pico()` command which is usually equal to `edit(..., editor="nano")`. nano editor is usually available also through the macOS terminal.

How to save the results

Beginners in R simply copy results of the work (like outputs from statistical tests) from the R console into some text file. This is enough if you are the beginner. Earlier or later, however, it becomes necessary to save larger objects (like data frames):

(File `trees.txt`, which is made from the internal `trees` data frame, will be written into the working directory.)

Please be really careful with `write.table()` as R is perfectly silent if the file with the same name `trees.txt` is already here. Instead of giving you any warning, it simply overwrites it!

By the way, “internal data” means that it is accessible from inside R directly, without preliminary loading. You may want to check which internal data is available with command `data()`.

While a `scan()` is a single-vector variant of `read.table()`, `write()` command is the single-vector variant of `write.table()`.

It is now a good time to speak about file name conventions in this book. We highly recommend to follow these simply rules:

1. Use only lowercase English letters, numbers and underscore for the file and directory names (and also dot, but only to separate file extension).
2. Do not use uppercase letters, spaces and other symbols!
3. Make your names short, preferably shorter than 15–20 symbols.
4. For R command (script) files, use extension `*.r`

By the way, for the comfortable work in R, it is strongly recommended to *change those options of your operating system which allow it to hide file extensions*. On macOS, go to Finder preferences, choose Advanced tab and *select* the appropriate box. On Windows, click View tab in File Explorer, choose Options, then View again, *unselect* appropriate box and apply this to all folders. Linux, as a rule, does not hide file extensions.

But what if we need to write into the external file *our results* (like the output from statistical test)? There is the `sink()` command: (Here the string “[1] 4” will be written to the external file.),

We specified `split=TRUE` argument because we wanted to see the result on the screen. Specify also `append=TRUE` if you want to *add* output to the existing file. To stop sinking, use `sink()` without arguments. Be sure that you always close `sink()`!

There are many tools and external packages which enhance R to behave like full-featured *report system* which is not only calculates something for you but also helps you to write the results. One of the simplest is Rresults shell script (<http://ashipunov.info/shipunov/r>) which works on macOS and Linux. The appendix of the book explains *Sweave* system. There are also *knitr* and much more.

History and scripts

To see what you typed during the current R session, run `history()`^[11]:

If you want to *save your history of commands*, use `savehistory()` with the appropriate file name (in quotes) as argument `(^[{12}])`.

While you work with this book, it is a good idea to use `savehistory()` and save all commands from each R session in the file named, saying, by the date (like `20170116.r`) and store this file in your working folder.

To do that on macOS, use menu R -> Preferences -> Startup -> History, uncheck Read history file on startup and enter the name of today’s history file. When you close R, file will appear in your working directory.

To save all objects in the binary file, type `save.image()`. You may want to use it if, for example, you are experimenting with R.

R allows to create *scripts* which might be run later to *reproduce* your work. Actually, R scripts could be written in any text editor^[13].

In the appendix, there is much more about R scripts, but the following will help you to create your own first one:

1. Open the text editor, or just type `file.edit("hello.r")`^[14]
2. Write there the string `print("Hello, world!")`
3. Save the file under `hello.r` name *in your working directory*
4. Call it from R using the command `source("hello.r")`
5. ... and you will see `[1] "Hello, world!"` in R console as if you typed it.

(In fact, you can even type in the script `"Hello world!"` without `print()`, R will understand what to do.)

Then, every time you add any R command to the `hello.r`, you will see more and more output. Try it.

To see input (commands) and output (results) together, type `source("hello.r", echo=TRUE)`.

Scripting is the “killer feature” of R. If all your data files are in place, and the R script is made, you may easily return to your calculations years later! Moreover, others can do exactly the same with your data and therefore your research becomes fully reproducible. Even more, if you find that your data must be changed, you run the same script and it will output results which take all changes into account.

Command `source()` allows to load commands not only from local file but also from Internet. You only need to replace file name with URL.

References

1. By the way, if you want the Euler number, *e*, type `exp(1)`.

2. And also like editor which is embedded into R for Windows or into R macOS GUI, or the editor from rite R package, but not office software like MS Word or Excel!
 3. Yet another possibility is to set working directory in preferences (this is quite different between operating systems) but this is not the best solution because you might (and likely will) want different working directories for different tasks.
 4. There is rio package which can determine the structure of data.
 5. Again, download it from Internet to data subdirectory first. Alternatively, replace subdirectory with URL and load it into R directly—of course, after you check the structure.
 6. On macOS, type Enter twice.
 7. With commands dput() and dget(), R also saves and loads textual representations of objects.
 8. This is a bit similar to the joke about mathematician who, in order to boil the kettle full with water, would empty it first and therefore *reduce the problem to one which was already solved!*
 9. If, by chance, it started and you have no idea how to quit, press uppercase ZQ.
 10. Within nano, use Ctrl+O to save your edits and Ctrl+X to exit.
 11. Does not work on graphical macOS.
 12. Under graphical macOS, this command is not accessible, and you need to use application menu.
 13. You can also use savehistory() command to make a “starter” script.
 14. On Windows and macOS, this will open internal editor; on Linux, it is better to set editor option manually, e.g., file.edit("hello.r", editor="geany").
-

2.7: R and Data is shared under a [Public Domain](#) license and was authored, remixed, and/or curated by LibreTexts.