

Lecture 3: R Basics

Peng Chap 4-6

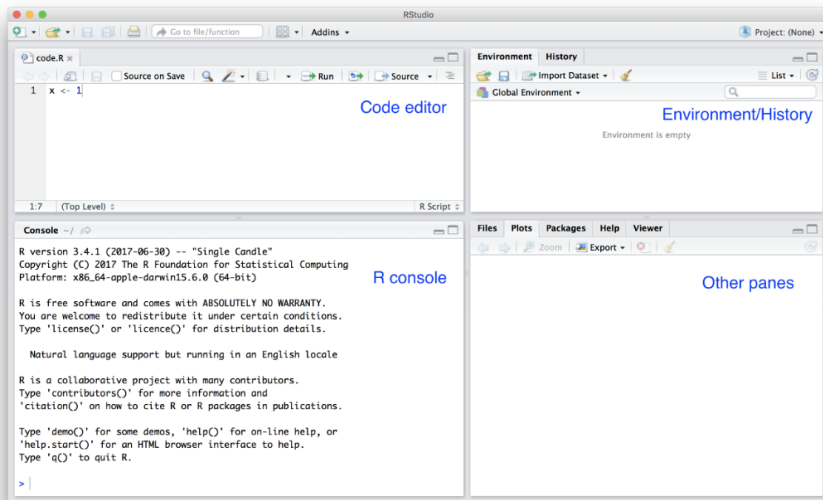
Ailin Zhang

Agenda

- R Studio
- Basic Operations and Data Type
- Input/ Output

Quick Survey: Have you installed RStudio?

RStudio Interface



RStudio Basics

- Key binding: to start a new script, Ctrl+Shift+N on Windows and command+shift+N on the Mac
- RStudio IDE cheatsheet: Help-> Cheatsheets -> RStudio IDE cheatsheet
- Load Libraries: `library()`
- Install packages:
Type in the console: `install.packages("name of the package")`

```
install.packages("tidyverse")  
# install more than one package  
install.packages(c("tidyverse", "ggplot2"))  
# check all installed packages  
installed.packages()
```

Basics: value assignment

We use assignment operator: `<-`

```
x <- 1    # Input  
x = 1  
msg <- "hello"
```

Question: Can we use `<-` and `=` interchangeably?

```
# Equal sign specifying parameters in function  
mean(a, na.rm=FALSE)  
# Creating a function  
sqroot <- function(n) sqrt(1:n)
```

Now, to check your understanding, what will happen to following lines?

```
system.time(result <- sqroot(1000))  
system.time(result = sqroot(1000))
```

Printing

```
x <- 5  ## nothing printed  
x       ## auto-printing occurs
```

```
## [1] 5
```

```
print(x)  ## explicit printing
```

```
## [1] 5
```

- Auto-print: easier for interactive work.
- Explicit-print: more convenient when writing scripts, functions, or longer programs.

R Objects

R has five basic objects (Everything you see in R!):

- character
- numeric (real numbers)
 - `L` suffix to specify integer: `1L`
 - `Inf` represents infinity, `1 / Inf` is 0.
 - `NaN`: undefined value (“not a number”) or missing value.
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector.

- A vector can **only contain objects of the same class**.
- Empty vectors can be created with the `vector()` function.

```
x <- vector("numeric", length = 10)
```

Vector

The `c()` function can be used to create vectors of objects by concatenating things together.

```
x <- c(0.5, 0.6)      ## numeric
x <- c(TRUE, FALSE)   ## logical
x <- c(T, F)           ## logical
x <- c("a", "b", "c") ## character
x <- 9:29               ## integer
x <- c(1+0i, 2+4i)     ## complex
```

Question: what happens to the following code?

```
y <- c(1.7, "a")
```

```
y
```

```
## [1] "1.7" "a"
```


Coercion

When different objects are mixed in a vector, **coercion** occurs so that every element in the vector is of the same class.

In the example above, we see the effect of implicit coercion. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion.

Sometimes, it could violate your expectation.

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
as.numeric(x)
```

```
## Warning: imaginary parts discarded in coercion
```

```
## [1] 1 2
```

```
as.logical(x)
```

Attribute

- Attributes are metadata for the object (to describe the object). Some examples of R object attributes are
 - names, dimnames
 - dimensions (e.g. matrices, arrays)
 - class (e.g. integer, numeric)
 - length
 - other user-defined attributes/metadata
- Attributes of an object (if any) can be accessed using the `attributes()` function.
- Not all R objects contain attributes: `attributes()` function returns `NULL`.

Matrices

There are several options to create a matrix

```
m <- matrix(nrow = 2, ncol = 3)
m
```

```
##           [,1] [,2] [,3]
## [1,]      NA  NA   NA
## [2,]      NA  NA   NA
```

```
dim(m)
```

```
## [1] 2 3
```

```
attributes(m)
```

```
## $dim
```

```
## [1] 2 3
```

Matrices

Constructed from vector: start from upper left corner, column-wise.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Or create directly from vectors by adding a dimension attribute.

```
m <- 1:10
dim(m) <- c(2, 5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Matrices

Column-binding or row-binding with the `cbind()` and `rbind()` functions.

```
x <- 1:3  
y <- 10:12  
cbind(x, y)
```

```
##      x  y  
## [1,] 1 10  
## [2,] 2 11  
## [3,] 3 12
```

```
rbind(x, y)
```

```
##  [,1] [,2] [,3]  
## x    1    2    3  
## y   10   11   12
```

Missing Values

- Missing values are denoted by `NA` or `NaN`.
- `is.na()` is used to test objects if they are NA
- `is.nan()` is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc. (More general)
- A NaN value is also NA but the converse is not true

Question: What is the output of following code?

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
is.nan(x)
```

Search for documentation

If you ever want to use a new function and would like to check its documentation

- `?pattern`
- `??pattern`
- `help.search(pattern)`

Reading Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file source, for reading in R code files
- `dget`, for reading in R code files
- `load`, for reading in saved workspaces.
- `unserialize`, for reading single R objects in binary form

For small to moderately sized datasets, you can usually call `read.table` without specifying any other arguments.

```
data <- read.table("foo.txt")
```

- R automatically figures out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.

Working Directory

If we get an error message during reading that says “No such file or directory”.

This is because we haven't told R where the file is located, which can be done by providing the complete path to the file or by setting the working directory to the folder the data file is located.

Set the Working Directory (1)

The screenshot shows the RStudio interface with the following components:

- Source Editor:** Contains R code for reading data. Lines 322-325 show a comment and a code block for negative indexes. Lines 327-338 show a section titled "## Reading Data" with a paragraph and a list of functions: `read.table()`, `read.csv()`, `readLines()`, `dget()`, `load()`, and `unserialize()`.
- Environment:** Shows "Global Environment" and "Environment is empty".
- Files Panel:** Displays the file explorer for the working directory. A red box highlights the "Files" tab and a yellow box highlights the instruction "Click on Files, and then navigate to the folder with data". The file list includes: `Books`, `Figs`, `L00.pdf`, `Lec 1 - Course Overview.Rmd`, `Lec 1 Course overview.pdf`, `Lec 1 Course overview.pptx`, `Lec1---Course-Overview.pdf`, `research proposal(2) copy.pdf`, and `Syllabus_STAT4130.pdf`.
- Console:** Shows the output of the R code, including "label: unnamed-chunk-6" and "label: unnamed-chunk-7 (with options)".

Set the Working Directory (2)

The screenshot shows the RStudio interface with the following components:

- Source Editor:** Contains R code for reading data. Lines 327-328 show a comment: `## Reading Data`. Lines 329-330 show a comment: `There are a few principal functions reading data into R.` Lines 331-336 show a list of functions: `read.table()`, `read.csv()`, `readLines()`, `dget()`, `load()`, and `unserialize()`. Line 337 shows a comment: `For small to moderately sized datasets, you can usually call`. Line 338 shows a comment: `read.table() without specifying any other arguments.`
- Environment Pane:** Shows "Environment is empty".
- Files Pane:** Shows a file tree with "Home" selected. A context menu is open, showing options like "Copy...", "Copy To...", "Move...", "Copy Folder Path to Clipboard", "Open Selected in Source Pane", "Open Each File in New Columns", "Set As Working Directory" (highlighted with a red box), "Go To Working Directory", "Synchronize Working Directory", "Open New Terminal Here", "Show Folder in New Window", and "Show Hidden Files".
- Console:** Shows the output of the R code. It includes a message: `Label: unnamed-chunk-6` and `Label: unnamed-chunk-7 (with options)`. It also shows a list of 1 file.

Example

```
data <- read.table("baby.csv", sep=";", header=TRUE)
head(data,n=3L)
```

```
##      Birth.Weight Gestational.Days Maternal.Age Maternal.Height
## 1             120             284             27             6
## 2             113             282             33             6
## 3             128             279             28             6
##      Maternal.Pregnancy.Weight Maternal.Smoker
## 1                      100             False
## 2                      135             False
## 3                      115              True
```

Question:

What if the dataset is too large?

Reading in Larger Datasets

Several Suggestions:

- Compare your RAM with memory required to store your dataset.
- Use the `colClasses` argument.

```
initial <- read.table("bigdata.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("bigdata.txt", colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage.

Exercise

- 1 Write your R code to read the first 100 rows from “baby.csv”, skip the headers, and assign it to variable ‘data’.
- 2 Check the class type of variable ‘data’, and print the total number of rows and columns for ‘data’. (Hint: nrow(), ncol())
- 3 Convert variable ‘data’ to matrix type and assign it to ‘dataM’. (Hint: data.matrix())

```
data <- read.table("baby.csv", sep=",", skip=1, nrows=100)
nrow(data)
```

```
## [1] 100
```

```
ncol(data)
```

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

```
dataM <- data.matrix(data)
dataM
```

Reading Data Summary

There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

For large dataset:

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- Operating systems, some limit the amount of memory a single process can access

Writing Data

There are analogous functions for writing data to files

- `write.table`, `write.csv`, for writing tabular data to text files (i.e. CSV)
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `serialize`, for converting an R object into a binary format.

Writing Data

```
m <- matrix(seq(1,100,5),4,5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   21   41   61   81
## [2,]    6   26   46   66   86
## [3,]   11   31   51   71   91
## [4,]   16   36   56   76   96
```

```
write.table(m,sep=' ',file="output.R")
m1 <- read.table("output.R",sep=' ')
m1
```

```
##    V1 V2 V3 V4 V5
## 1   1 21 41 61 81
## 2   6 26 46 66 86
## 3  11 31 51 71 91
```

Using the readr Package

The `readr` package is recently developed by Hadley Wickham to deal with reading in large flat files quickly.

```
library(readr)
read_csv(mtcars_path)
write_csv(mtcars, mtcars_path)
```

Interfaces to the Outside World

Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- file, opens a connection to a file
- gzfile, opens a connection to a file compressed with gzip
- bzfile, opens a connection to a file compressed with bzip2
- url, opens a connection to a webpage

Connect to text files

Connections to text files can be created with the `file()` function.

```
str(file)
```

```
## function (description = "", open = "", blocking = TRUE, enc  
##      raw = FALSE, method = getOption("url.method", "default"
```

The `open` argument allows for the following options:

- “r” open file in read only mode
- “w” open a file for writing (and initializing a new file)
- “a” open a file for appending
- “rb”, “wb”, “ab” reading, writing, or appending in binary mode

Example - 1

```
## Create a connection to 'foo.txt'  
con <- file("foo.txt")  
  
## Open connection to 'foo.txt' in read-only mode  
open(con, "r")  
  
## Read from the connection  
data <- read.csv(con)  
  
close(con)
```

Example - 2

Open connection to gz-compressed text file

```
con <- gzfile("words.gz")  
x <- readLines(con, 10)
```

The above example used the `gzfile()` function which is used to create a connection to files compressed using the gzip algorithm. There is a complementary function `writeLines()` that takes a character vector and writes each element of the vector one line at a time to a text file.

Example - 3

Open connection to url

```
con <- url("http://www.sjtu.edu.cn", "r")  
## Read the web page  
x <- readLines(con)  
## Print out the first few lines  
head(x)
```

Using URL connections can be useful:

- 1 producing a reproducible analysis
- 2 Opening a web browser and downloading a dataset by hand.

Of course, the code you write with connections may not be executable at a later date if things on the server side are changed or reorganized.

Your turn!

Please read chapter 3 of Introduction to Data Science (2020) by Rafael Irizarry. (<https://rafalab.github.io/dsbook>)

We will ask you to write r code for homework 1 (coming tonight!)