Programming in R

Nora Wickelmaier

Last modified: October 7, 2024

Programming resources

Learning statistics with R	https://learningstatisticswithr.com/book/
R for Data Science	https://r4ds.hadley.nz/
Advanced R	https://adv-r.hadley.nz/
Happy Git and GitHub for the useR	https://happygitwithr.com/
R Programming for Research	https://geanders.github.io/RProgrammingForResearch/
Building reproducible analytical pipelines with R	https://raps-with-r.dev/
Data Skills for Reproducible Science	https://psyteachr.github.io/msc-data-skills/

1 Style guidelines

Style guidelines in R

- R has no mandatory or commonly accepted style guide

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
  - https://google.github.io/styleguide/Rguide.html

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
    - https://google.github.io/styleguide/Rguide.html
    - https://style.tidyverse.org/

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
  - https://google.github.io/styleguide/Rguide.html
  - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not “create” your own rules (if you deviate, be consistent!)

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
    - https://google.github.io/styleguide/Rguide.html
    - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not "create" your own rules (if you deviate, be consistent!)
- A style guide helps with

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
    - https://google.github.io/styleguide/Rguide.html
    - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not “create” your own rules (if you deviate, be consistent!)
- A style guide helps with
    - Keeping code clean which is easier to read and interpret

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
  - https://google.github.io/styleguide/Rguide.html
  - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not “create” your own rules (if you deviate, be consistent!)
- A style guide helps with
  - Keeping code clean which is easier to read and interpret
  - Making it easier to catch and fix mistakes

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
  - https://google.github.io/styleguide/Rguide.html
  - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not “create” your own rules (if you deviate, be consistent!)
- A style guide helps with
  - Keeping code clean which is easier to read and interpret
  - Making it easier to catch and fix mistakes
  - Making it easier for others to follow and adapt your code

Style guidelines in R

- R has no mandatory or commonly accepted style guide
- However, Hadley Wickham and Google developed style guides which are now widely accepted
  - https://google.github.io/styleguide/Rguide.html
  - https://style.tidyverse.org/
- It is always a good idea to follow a style guide and not "create" your own rules (if you deviate, be consistent!)
- A style guide helps with
  - Keeping code clean which is easier to read and interpret
  - Making it easier to catch and fix mistakes
  - Making it easier for others to follow and adapt your code
  - Preventing possible problems, e. g., avoiding dots in function names

File names I

- File names should be meaningful and end in .R
- Avoid using special characters in file names
- Stick with numbers, letters, -, and _

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

File names II

- If files should be run in a particular order, prefix them with numbers
- If it seems likely you'll have more than 10 files, left pad with zero

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

- If you later realize that you missed some steps, it's tempting to use 02a, 02b, etc.
- However, it is generally better to bite the bullet and rename all files

Object names I

- Variable and function names should use only lowercase letters, numbers, and _
- Use underscores (_) (so called snake case) to separate words within a name

```
# Good
day_one
day_1

# Bad
DayOne
dayone
```

Object names II

- Generally, variable names should be nouns and function names should be verbs
- Strive for names that are concise and meaningful

```
# Good
day_one

# Bad
first_day_of_the_month
djm1
```

Object names III

- Avoid re-using names of common functions and variables

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Spacing I

- Always put a space after a comma, never before

```
# Good
x[, 1]

# Bad
x[,1]
x[ ,1]
x[ , 1]
```

Spacing II

- Do not put spaces inside or outside parentheses for regular function calls

```
# Good
mean(x, na.rm = TRUE)

# Bad
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

Spacing III

- Place a space before and after `()` when used with `if`, `for`, or `while`

```
# Good
if (debug) {
  show(x)
}

# Bad
if(debug){
  show(x)
}
```

Spacing IV

- Place a space after () used for function arguments

```
# Good
function(x) {}

# Bad
function (x) {}
function(x){}
```

Spacing V

- Most infix operators (==, +, -, <-, etc.) should always be surrounded by spaces

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

Spacing VI

- There are a few exceptions, which should never be surrounded by spaces: `::`, `:::`, `$`, `@`, `[`, `[[`, `?`, `^`, and `:`

```
# Good
sqrt(x^2 + y^2)
df$z
x <- 1:10
package?stats
?mean

# Bad
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10
package ? stats
? mean
```

Spacing VII

- Adding extra spaces is ok if it improves alignment of = or <-

```
# Good
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)

# Also fine
list(
  total = a + b + c,
  mean = (a + b + c) / n
)
```

2 Script organisation

Script header

- It can be very helpful to have some general information right at the top when opening a script

```
# 01_preprocessing.R
#
# Cleaning up toy data set (Methods Seminar SS2024)
#
# Input:  rawdata/RDM_MS_SS2024_download_2024-06-07.csv
# Output: processed/data_rdm-ms-ss2024_cleaned.csv
#         processed/data_rdm-ms-ss2024_cleaned.RData
#
# Created: 2024-06-03, NW
```

- These metadata help you remember faster what you did
- Might not be necessary when using consistent version control (but does not hurt either)

Line length

Keep lines to 80 characters or less!

```
# Good
my_df <- data.frame(n = 1:3,
                    letter = c("a", "b", "c"),
                    cap_letter = c("A", "B", "C"))

# Bad
my_df <- data.frame(n = 1:3, letter = c("a", "b", "c"), cap_letter = c("A"
```

- Ensures that your code is formatted in a way that you can see all of the code without scrolling horizontally
- To set your script pane to be limited to 80 characters, go to `RStudio -> Preferences -> Code -> Display` and set "Margin Column" to 80

File organisation I

- Try to write scripts that are concerned with one (major) task
- If you can find a name, that captures the content, it is usually a good way to start
- Some (random) examples

```
download-data.R
data-cleaning.R
cluster_analysis_exp1.R
visualization_logistic-model.R
anova_h1.R
```

File organisation II

- Export data sets for new scripts (do not make yourself run all scripts up to script 5 each time, just because you need the data in a certain format)

```
# Interoperable
write.table(dat,
            file = "data_exp1_cleaned.csv",
            sep = ";",
            quote = FALSE,
            row.names = FALSE)

# Preserve order of factor levels, date formats, etc.
save(dat, file = "data_exp1_cleaned.RData")
```

Internal structure I

- Use commented lines with – or = to break your file up into chunks
- Load additional packages at the beginning of the script

```
library(lme4)
library(sjPlot)

# Load data ----------------------------

# Plot data ----------------------------
```

Internal structure II

- If you load several packages, be aware that the order of loading matters!
- If you use only one or two functions from a package, get the function with `::` instead of loading the whole package

```
library(lme4)
...

# Fit mixed-effects model to test Hypothesis 1
lme1 <- lmer(Reaction ~ Days + (Days | Subject), sleepstudy)
summary(lme1)
sjPlot::tab_model(lme1)
```

Internal structure III

- Group related pieces of code together
- Separate blocks of code by empty spaces

```
# Load data
library(faraway)
data(nepali)

# Relabel sex variable
nepali$sex <- factor(nepali$sex,
                     levels = c(1, 2),
                     labels = c("Male", "Female"))
```

3 Writing functions

Functions and arguments

- Functions in R consist of
  - a name
  - a pair of brackets
  - the arguments (none, one, or more)
  - a return value (visible, invisible, `NULL`)
- Arguments are passed
  - either without name (in the defined order)
    → positional matching
  - or with name (in arbitrary order)
    → keyword matching
- Even if no arguments are passed, the brackets need to be written, e. g., `ls()`, `dir()`, `getwd()`
- Entering only the name of a function without brackets will display the R code of that function

Writing functions

- Let us implement a simple function ourselves
- A function that implements a two-sample t test

$$T = \frac{\bar{x} - \bar{y}}{\sqrt{\hat{\sigma}^2 \, \left(\frac{1}{n} + \frac{1}{m}\right)}} = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{(n-1)\, s_x^2 + (m-1)\, s_y^2}{n+m-2} \cdot \left(\frac{1}{n} + \frac{1}{m}\right)}}$$

with

$$T \sim t(n + m - 2)$$

Writing functions

```
# Example: a handmade t test function
twosam <- function(y1, y2){                   # definition
  n1  <- length(y1); n2  <- length(y2)    # body
  yb1 <- mean(y1);   yb2 <- mean(y2)
  s1  <- var(y1);    s2  <- var(y2)
  s   <- ((n1 - 1)*s1 + (n2 - 1)*s2)/(n1 + n2 - 2)
  tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
  tst             # return value, can also be a list
}

# Calling the function
tstat <- twosam(PlantGrowth$weight[PlantGrowth$group == "ctrl"],
                PlantGrowth$weight[PlantGrowth$group == "trt1"])
tstat
```

Named arguments

- If there is a function `fun1` defined by

```
fun1 <- function(data, data.frame, graph, limit) {
...
}
```

then the function may be invoked in several ways, for example

```
fun1(d, df, TRUE, 20)
fun1(d, df, graph = TRUE, limit = 20)
fun1(data = d, limit = 20, graph = TRUE, data.frame = df)
```

- All of them are equivalent (cf. positional matching and keyword matching)

Defaults

- In many cases, arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call

```
fun1 <- function(data, data.frame, graph = TRUE, limit = 20) {
  ...
}
```

- It could be called as

```
ans <- fun1(d, df)
```

which is now equivalent to the three cases above, or as

```
ans <- fun1(d, df, limit = 10)
```

which changes one of the defaults

Exercise

- Write a function in R that cumulatively sums up the values of vector $\mathbf{x} = (1\ 2\ 3\ 4 \ldots 20)'$
- The result should look like: $\mathbf{y} = (1\ 3\ 6\ 10 \ldots 210)'$

4 Conditional execution

Conditional execution

- When programming, a distinction of cases is often necessary for
  - checking of arguments
  - return of error messages
  - interrupting a running process
  - case distinction, e. g., in mathematical expressions
- Conditional execution of code is available in R via

```
if(expr_1) {
  expr_2
} else {
  expr_3
}
```

where `expr_1` must evaluate to a single logical value

Conditional execution

```
x <- 5

# Example 1
if(!is.na(x)) y <- x^2 else stop("x is missing")

# Example 2
if(x == 5){   # in case x = 5:
  x <- x + 1  # add 1 to x and
  y <- 3      # set y to three
} else        # else:
  y <- 7      # set y to seven

# Example 3
if(x < 99) cat("x is smaller than 99\n")
```

Conditional execution

```
## Vectorized version with ifelse() function

# Example 1
ifelse(x == c(5, 6), c("A1", "A2"), c("A3", "A4"))

# Example 2
x <- -2:2
ifelse(x < 0, -x, x)
```

Exercise

- Implement the following function in R:

$$f(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

5 Loops

Loops

- Loops are necessary to execute repeating commands
- Especially for simulations, loops are often used
- In this case, the same functions or commands are executed for different random numbers
- There are `for()` and `while()` loops for repeated execution
- The most simple “loop” is `replicate()`

Loops

```
## Example for central limit theorem

y <- runif(100)     # Draw random numbers
hist(y)

x <- replicate(1000, mean(runif(100)))
hist(x)
```

Loops

```
i <- 0
repeat{
  i <- i + 1            # Add 1 to i
  if (i == 3) break     # Stop if i = 3
}

while (i > 1) {
  i <- i - 1            # As long as i > 1, subtract 1
}
```

Loops

```
x <- c(3, 6, 4, 8, 0)        # Vector of length 5
for(i in x)
  print(sqrt(i))

for(i in seq_along(x))       # Same using indices
  print(sqrt(x[i]))

for(i in seq_along(x)){      # For all i [1,2,3,4,5]'
  x[i] <- x[i]^2             # square ith element of x
  print(x[i])                # and show it on monitor
}

x^2                          # BETTER
```

Exercise

- Create a vector $\mathbf{x} = (3\ 5\ 7\ 9\ 11\ 13\ 15\ 17)'$ with a for-loop
  Tip: Use the formula $n \cdot 2 + 1$
- Implement two different methods:
  1. Allocate memory: Start with a vector of zeros and the correct length and replace its elements iteratively
  2. Growing: Start with a `NULL` object and iteratively add new results

  Tip: The first method is more efficient, especially for long vectors

6 Avoiding loops

Avoiding loops

- The `apply()` family of functions may be used in many places where in traditional languages loops are employed
- Using vector based alternatives is usually much faster in R

Matrices and arrays:	`apply()`
Data frames, lists and vectors:	`lapply()` and `sapply()`
Group-wise calculations:	`tapply()`

`apply()`

- `apply()` is used to work vector-wise on matrices or arrays
- Appropriate functions can be applied to the columns or rows of a matrix or array without explicitly writing code for a loop

apply()

```r
X <- matrix(c(4, 7, 3, 8, 9, 2, 5, 6, 2, 3, 2, 4), nrow = 3, ncol = 4)

# Calculate row maxima
res <- numeric(nrow(X))
for(i in 1:nrow(X)){
  res[i] <- max(X[i,])
}

# or:
apply(X, 1, max)  # Maximum for each row
apply(X, 2, max)  # Maximum for each column
```

`lapply()` and `sapply()`

lapply Using the function `lapply()` (`l` because the value returned is a list), another appropriate function can be applied element-wise to other objects, e. g., data frames, lists, or simply vectors

→ The resulting list has as many elements as the original object to which the function is applied

sapply Analogously, the function `sapply()` (`s` for simplify) works like `lapply()` with the exception that it tries to simplify the value it returns

→ It might become a vector or a matrix

lapply() and sapply()

```
L <- list(x = 1:10, y = 1:5 + 0i)
lapply(L, mean)        # Keep list with data type
sapply(L, mean)        # Create vector, same data type

sapply(iris, class)    # Work column wise on data frame
```

`tapply()`

- `tapply()` (`t` for table) may be used to do group-wise calculations on vectors
- Frequently, it is employed to calculate group-wise means

tapply()

```
data(Oats, package = "nlme")
with(Oats, tapply(yield, list(Block, Variety), mean))

data(warpbreaks)
tapply(warpbreaks$breaks, warpbreaks$tension, sum)
tapply(warpbreaks$breaks, warpbreaks[ , -1], mean)
```

Exercise

- Load the iris data set into R using `data(iris)`
- Write a for-loop to calculate the means for the dependent variables (columns 1 to 4)
- Think of as many vector based alternatives as possible to avoid this loop and calculate the column means

7 Random number generation

Random number generation

- Most distributions that R handles have four functions
- There is a root name, e. g., the root name for the normal distribution is `norm`
- This root is prefixed by one of the letters `p`, `q`, `d`, `r`

`p`	probability: the cumulative distribution function (CDF)
`q`	quantile: the inverse CDF
`d`	density: the probability (density) function (PDF)
`r`	random: a random variable having the specified distribution

Random number generation

- See `?Distributions` for a list of distributions or the CRAN task view https://cran.r-project.org/view=Distributions
- The random number generator in R is *seeded*: Upon restart of R, new random numbers are generated
- To replicate the results of a simulation, the seed (starting value) can be set explicitly

Random number generation

```
# Examples
rnorm(10)      # Draw from standard normal distribution
rpois(10, 1)   # Draw from Poisson distribution

# Sampling with or without replacement from a vector
sample(1:5, size = 10, replace = TRUE)

# Set seed
set.seed(1223)  # On each run, random numbers will be identical
runif(3)
```

Exercise

- Create a data frame with four variables and 20 observations:
  1. $X \sim N(\mu = 100, \sigma^2 = 15^2)$
  2. $Y \sim Bin(n = 10, p = 0.2)$
  3. $Z \sim Pois(\lambda = 1)$
  4. $S = X + Y + Z$

8 Data frames

The fundamental data structure: data frames

- Data frames are lists that consist of vectors and factors of equal length
- The rows in a data frame refer to one unit (observation or subject)
- For longitudinal data they can be reshaped between the long and the wide data format

The fundamental data structure: data frames

```r
# Creating a data frame
id <- factor(paste("s", 1:6, sep = ""))
weight <- c(60, 72, 57, 90, 95, 72)
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
dat <- data.frame(id, weight, height)

# Variables in a data frame are extracted by $
dat$weight
```

Working with data frames

```
# Frequently used functions (not only) for data frames

dim(dat)        # Show number of rows and columns

names(dat)      # Variable names

View(dat)       # Open data viewer; might be of little
                # help for large data sets

plot(dat)       # Pairwise plots

str(dat)        # Show variables of dat

summary(dat)    # Descriptive statistics
```

Indexing variables

```
weight[4]            # 4th element
weight[4] <- 92      # Change 4th element
weight[c(1, 2, 6)]   # Elements 1, 2, 6
weight[1:5]          # Elements 1 to 5
weight[-3]           # Without element 3
# See ?Extract

# Indices may be logical.
weight[weight > 60]
weight[weight > 60 & weight < 80]
height[weight > 60 & weight < 80]
```

Logical expressions may contain AND `&`, OR `|`, EQUAL `==`, NOT EQUAL `!=`, and `<`, `<=`, `>`, `>=`, `%in%`.

Indexing data frames

```
dat[3, 2]     # 3rd row, 2nd column
dat[1:4, ]    # Rows 1 to 4, all columns
dat[, 3]      # All rows, 3rd column

dat[dat$id == "s2", ]    # All observations of s2
dat[dat$weight > 60, ]   # All observations above 60kg
```

Exercise

- Create a data frame with two independent variables: *Hand* with levels "right" and "left" and *Condition* with levels 1, 2, 3, 4, and 5
- Simulate reaction times for 50 subjects; assume reaction time is normally distributed with $RT \sim N(\mu = 400, \sigma^2 = 625)$
- There are 10 subjects in each condition
- Use functions `str()` and `summary()` on your data frame:
  What does the output tell you?

References

Anderson, B., Severson, R., & Good, N. (2023). *R programming for research*. Colorado State University, ERHS 535. https://geanders.github.io/RProgrammingForResearch/

Ligges, U. (2008). *Programmieren mit R*. Springer-Verlag.

Venables, W. N., Smith, D. M., R Development Core Team, et al. (2022). An introduction to R. https://cran.r-project.org/doc/manuals/r-release/R-intro.html

Wickham, H. (n.d.). The tidyverse style guide. https://style.tidyverse.org/