R Refresher

Content

1. Obtain and install R and RStudio
2. Getting to know RStudio
3. Learning the building blocks of the R language

1. Obtain and install R and RStudio

R

R is an open source (interpreted) programming language for statistical computing and graphics

- ranks among 10 most popular languages
- under continuous development
- thousands of user-created packages, which allow for canned specialized techniques
- large and active online community that has virtually every problem solved for you

Install R

To install R

- visit https://cran.r-project.org/ in your browser
- select the link that matches your operating system
- follow installation instructions, go for R version 3.6.0

RStudio

RStudio is an open source integrated development environment (IDE) which includes

- a text editor to write programs
- a console where programs are executed
- two multipurpose viewers showing graphs, the global environment section, etc.

Install RStudio

To install RStudio

- visit https://www.rstudio.com/products/rstudio/download/#download
- under “Installers” select the link that matches your operating system
- follow installation instructions

2. Getting to know R Studio

Basic setup

Within RStudio

- from the menu on top select ‘Tools’ > ‘Global Options’
- on the left panel, select ‘Code’
- under ‘Editing’ make sure ‘Ctrl+Enter executes’ defaults to ‘Current line’
- under ‘Saving’ make sure that ‘Default text encoding’ shows ‘UTF-8’
- on the left panel, select ‘Pane layout’
- make sure that the top left form shows ‘Source’ and the top right shows ‘Console’
- you can customize the bottom left and right forms as you like, I prefer to have the ‘Environment’, ‘History’, ‘Files’, ‘Plots’, ‘Packages’, and ‘Help’ in the lower right pane and the rest, which I rarely use, in the lower left
- click ‘Apply’ and then ‘OK’
- Otherwise, the defaults should be fine but you can further explore the global options to customize the ‘Appearance’ of RStudio, how RStudio highlights code snippets for you, autocompletes functions, and so on.

Source

The source pane is essentially a text editor where you document your R code

- on top is a project manager, which allows you to switch between R scripts
- # is used to place comments in code
- Note that R is case sensitive
- use CTRL+S to save your script
- use CTRL+ENTER to execute your script
- use CTRL+F to search and replace within your script
- More shortcuts? Try ALT+SHIFT+K

Console

The console pane is where your script is executed and output is shown

- R is ready when the console offers >
- a red stop sign indicates that R is computing, click it to cancel execution or hit ESC
- on top you see your current working directory, to change your working directory execute the following line of code with your directory typed inbetween quotes, note the use of forward slashes

```
setwd("FILE/PATH/TO/YOUR/DIRECTORY")
```

- input is incomplete if R answers with + (likely forgotten a ‘)’ or ‘]’)
- recycle previous commands by using the arrow keys
- use CTRL+1 to switch to ‘Source’ and CTRL+2 to switch back to ‘Console’

The multipurpose panels

The multipurpose panels offer different functionalities depending on how you customized them

- the ‘Environment’ is a workspace viewer that lists objects (data, values, functions) you created and provides detailed information, it also allows to manually import data or a previously saved workspace
- the ‘History’ lists every single command executed in the current session
- ‘Plots’ displays graphs you generated
- ‘Packages’ shows installed packages with a brief description and allows you to manually load, detach, and install additional packages
- ‘Help’ provides detailed documentation of functions

3. Learning the building blocks of the R language

Control abstraction, i.e., how to tell R what to do

Arithmetic operations

```r
5 + 3 # addition
#> [1] 8
5 - 3 # subtraction
#> [1] 2
5 ^ 3 # exponentiation
#> [1] 125
5 ** 3 # exponentiation
#> [1] 125
5 * 3 # multiplication
#> [1] 15
5 / 3 # division
#> [1] 1.666667
5 * (10 - 3) # use of brackets
#> [1] 35
10 %% 3 # modulo, remainder of division
#> [1] 1
10 %/% 3 # integer divide
#> [1] 3
```

Relational operations

```r
5 > 3 # greater than
#> [1] TRUE
5 < 3 # less than
#> [1] FALSE
5 <= 3 # weakly less than
#> [1] FALSE
5 >= 3 # weakly greater than
#> [1] TRUE
```

```
5 == 3 # equals
#> [1] FALSE
5 != 3 # unequal
#> [1] TRUE
```

Assignment

R stores information as an object (in the environment) with a name of your choice. An object name cannot begin with a number, spaces, or special characters that have special meaning in R. Avoid function names. `<-` is the assignment operator, `=` works, too, but is considered bad practice.

```
result <- 5 + 3 # assign
result # call object, note that calling 'Result' will throw an error as R is case
        sensitive
#> [1] 8
result <- 5 ^ 3 # overwrite object
result
#> [1] 125
results <- c(5 + 3, 5 - 3, result) # assigning multiple values using 'c()' -
        concatenate
results
#> [1]   8   2 125
```

Logical operations

```
5 & 3 %in% results # logical conjunction (and)
#> [1] FALSE
5 == 2 | 3 == 2 # logical disjunction (or)
#> [1] FALSE
!3 %in% results # logical negation
#> [1] TRUE
```

Function calls

A function takes one or multiple inputs (called 'arguments') within brackets and produces an output. To learn about a function and its arguments type `?` in front of it and check the respective multipurpose panel for the function documentation.

```
mean(x = results) # arithmetic mean
#> [1] 45
base::mean(x = results) # to avoid conflict with other packages, specify the function
        source, i.e., the package, as 'source::', usually not necessary
#> [1] 45
sqrt(x = result) # square root
#> [1] 11.18034
sum(x = results) # sum of elements
```

```
#> [1] 135
seq(from = 1, to = 10) # sequence
#>  [1]  1  2  3  4  5  6  7  8  9 10
values <- seq(from = 1, to = 10, by = 2)
rep(x = result, times = 3) # replication of elements
#> [1] 125 125 125
rep(x = results, each = 3)
#> [1]   8   8   8   2   2   2 125 125 125
print(values) # print values
#> [1] 1 3 5 7 9
print(c(values, result))
#> [1]   1   3   5   7   9 125
print(mean(results)) # nested function calls
#> [1] 45
```

Subsetting operations

R provides three subsetting operators, `[`, `[[`, and `$` to extract or replace parts of an object. You will learn about these in detail below together with how to access different data structures. For now, just consider the `[` operator. Use `[]` to access, i.e., index, elements by position from the objects created thus far.

```
results[3] # access the third element in results
#> [1] 125
results # 125 is the third element in results
#> [1]   8   2 125
```

Control structures

Control structures allow you to control the flow of execution in a script, we distinguish conditional execution, loops, and conditional jumps. You can use all these structures to build your own functions, too (not covered here).

Conditional execution with `if` - `if (condition) {do}`

```
if ( 3 %in% c(1,2,3)) {
  print("There is 3")
}
#> [1] "There is 3"

print(result)
#> [1] 125
print(results)
#> [1]   8   2 125

if (result %in% results) {
  cat(result, "is in results")
}
```

```
#> 125 is in results
```

Conditional execution with `if` and `else` - `if (condition) {do} else {do}`

```
some_number <- sample(x = 1:20, size = 1)

if (some_number <= 10) {
  cat(some_number, " is less than 10")
} else {
  cat(some_number, " is greater than 10")
}
#> 18  is greater than 10
```

Conditional execution with vectorized `ifelse` - `ifelse(condition, if met do, else do)`

```
values <- sample(x = 1:20)
ifelse(values >= 10, TRUE, FALSE)
#>  [1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
#> [13] FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

`for` loop for iterative tasks - `for (element in sequence of elements) {do}`

```
for (i in 1:length(values)) {
  print(values[i])
}
#> [1] 13
#> [1] 5
#> [1] 15
#> [1] 16
#> [1] 20
#> [1] 8
#> [1] 6
#> [1] 9
#> [1] 11
#> [1] 3
#> [1] 19
#> [1] 4
#> [1] 7
#> [1] 1
#> [1] 17
#> [1] 12
#> [1] 2
#> [1] 10
#> [1] 14
#> [1] 18
```

Note how ‘i’ appears in the global environment, this is a side effect of using control structures outside of functions - the state of the program is changed, i.e., the global environment is affected. Mind that this can have unanticipated consequences. `for` loop with conditional execution

```
for (i in 1:length(values)) {
  if (values[i] >= 10) {
    print(TRUE)
  } else {
    print(FALSE)
  }
}
#> [1] TRUE
#> [1] FALSE
#> [1] TRUE
#> [1] TRUE
#> [1] TRUE
#> [1] FALSE
#> [1] FALSE
#> [1] FALSE
#> [1] TRUE
#> [1] FALSE
#> [1] TRUE
#> [1] FALSE
#> [1] FALSE
#> [1] FALSE
#> [1] TRUE
#> [1] TRUE
#> [1] FALSE
#> [1] TRUE
#> [1] TRUE
#> [1] TRUE
```

`while` loop - `while (condition is met) {do}`

```
while (result < 200) {
  print(result)
  result <- result + 5
}
#> [1] 125
#> [1] 130
#> [1] 135
#> [1] 140
#> [1] 145
#> [1] 150
#> [1] 155
#> [1] 160
#> [1] 165
#> [1] 170
#> [1] 175
#> [1] 180
#> [1] 185
#> [1] 190
```

```
#> [1] 195
```

Again, note how the object “result” is altered in the global environment.

Conditional jump with `next` - skips processing element further and begins with next iteration, not required in this example but useful for nested loops or exception handling

```
for(i in 1:length(values)) {
  if(values[i] %% 2 == 0) {
    next
  }
  values[i] <- 0
}
```

Conditional jump with `break` - stops execution of loop upon condition

```
values
#>  [1]  0  0  0 16 20  8  6  0  0  0  0  4  0  0  0 12  2 10 14 18
for(i in 1:length(values)) {
  print(i)
  values[i] <- values[i] + 1
  if(all(values != 0)) {
    break
  }
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
#> [1] 10
#> [1] 11
#> [1] 12
#> [1] 13
#> [1] 14
#> [1] 15
values
#>  [1]  1  1  1 17 21  9  7  1  1  1  1  5  1  1  1 12  2 10 14 18
```

Data types

To make the best of the R language, you’ll need a strong understanding of the basic data types and data structures and how to operate on them. Everything in R is an object.

R has 6 basic data types:

- character
- numeric (real or decimal)
- integer
- boolean
- complex

Elements of these data types may be combined to form data structures, such as atomic vectors. When we call a vector atomic, we mean that the vector only holds data of a single data type. Below are examples of atomic character vectors, numeric vectors, integer vectors, etc.

- character: “a”, “swc”
- numeric: 2, 15.5
- integer: 2L (the L tells R to store this as an integer)
- boolean: TRUE, FALSE
- complex: 1+4i (complex numbers with real and imaginary parts)

R provides many functions to examine features of vectors and other objects, for example

- class() - what kind of object is it (high-level)?
- typeof() - what is the object’s data type (low-level)?
- length() - how long is it? What about two dimensional objects?
- attributes() - does it have any metadata?

Boolean

The boolean data type represents logical values, in R `TRUE` or `FALSE`, alternatively `T` or `F`. Matching, comparison, and set operations often evaluate to logical values.

```
boolean <- TRUE
boolean
#> [1] TRUE
typeof(boolean)
#> [1] "logical"
boolean <- F
boolean
#> [1] FALSE
typeof(boolean)
#> [1] "logical"
typeof(1 == 2) # comparison operation
#> [1] "logical"
results %in% values # matching operation
#> [1] FALSE  TRUE FALSE
```

Integer

The integer data type represents whole numbers. This requires less storage capacity. If not made explicit by appending ‘L’ to a number, the number is autocoerced to type ‘numeric’ in R.

```
whole <- c(2L, 14L, 36L)
```

```
whole
#> [1]  2 14 36
typeof(whole)
#> [1] "integer"
```

Numeric

The numeric data type represents real and decimal numbers which require more storage capacity as they are stored as double precision floating point numbers (consists of sign, exponent, and mantisse).

```
decimal <- c(2.14, 3, 1836.819120)
decimal
#> [1]    2.140    3.000 1836.819
typeof(decimal)
#> [1] "double"
class(decimal)
#> [1] "numeric"
```

Character

The character data type represents strings consisting of no, one, or more numbers or characters set between double quotes. Use single quotes within strings, encoding matters here, western standard is UTF-8.

```
string <- ("multilevel")
typeof(string)
#> [1] "character"
```

Type transformation

R supports strong typing, i.e., it imposes strict restrictions on valid operations `result + "5"` throws an error. To transform data types, use `as.logical()`, `as.integer()`, `as.numeric()`, and `as.character()`.

```
typeof(as.numeric("5"))
#> [1] "double"
result + as.numeric("5")
#> [1] 205
as.character(result)
#> [1] "200"
typeof(as.integer("2"))
#> [1] "integer"
```

Data structures

R has many data structures. Here I introduce four major data structures:

- vector
- matrix
- data frame
- list

Vectors

Vectors are homogenous, one dimensional arrays which represent a collection of information stored in a specific order. Vectors are accessed with the `[` operator.

```
result # a scalar, or a vector of length 1
#> [1] 200
values # a vector, a collection of elements
#>  [1]  1  1  1 17 21  9  7  1  1  1  1  5  1  1  1 12  2 10 14 18
log_vect <- c(TRUE, FALSE, T, F) # a logical vector
length(log_vect) # length of vector
#> [1] 4
str(log_vect) # structure of vector
#>  logi [1:4] TRUE FALSE TRUE FALSE
cha_vect <- c("a", "b", "c") # a character vector
str(cha_vect)
#>  chr [1:3] "a" "b" "c"
c(1, 2, "3", TRUE, 5) # coercion to most flexible type - character
#> [1] "1"    "2"    "3"    "TRUE" "5"
c(1, 2, FALSE, 5) # coercion to most flexible type - numeric
#> [1] 1 2 0 5
c(1, 2, NA, 3) # special values in a vector, NA - missing data
#> [1]  1  2 NA  3
results[3] # access third element
#> [1] 125
results[c(2,3)] # access second and third element
#> [1]   2 125
results[c(FALSE, TRUE, TRUE)] # same
#> [1]   2 125
results[3] <- 4 # replace
results[3] # now the third element is 4
#> [1] 4
results <- results[-3] # remove
results # now there is no third element anymore
#> [1] 8 2
results[results > 3] # access by using conditions
#> [1] 8
```

Matrices

Matrices are homogenous, two dimensional arrays implemented as vectors. Matrices are accessed with the [operator.

```
matrix_1 <- matrix(data = 1:6, nrow = 2, ncol = 3) # create a matrix with 'matrix()'
matrix_1
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
matrix_2 <- array(data = 1:6, dim = c(2, 3)) # or use 'array()' which is also used to
        construct multidimensional arrays (not covered here)
matrix_2
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
dim(matrix_1) # dimensions of a matrix, two rows, three columns
#> [1] 2 3
str(matrix_1) # structure of a matrix
#>  int [1:2, 1:3] 1 2 3 4 5 6
nrow(matrix_1) # number of rows, same as dim(matrix_1)[1]
#> [1] 2
ncol(matrix_1) # number of columns, same as dim(matrix_1)[2]
#> [1] 3
length(matrix_1) # number of rows times number of columns
#> [1] 6
# to combine matrices, use 'cbind()' and 'rbind()'
cbind(matrix_1, matrix_2) # add columns to a matrix
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    1    3    5    1    3    5
#> [2,]    2    4    6    2    4    6
rbind(matrix_1, matrix_2) # add rows to a matrix
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#> [3,]    1    3    5
#> [4,]    2    4    6
matrix_2[2, 3] # access using index with two positions [rows, columns], otherwise
        works same as for vectors
#> [1] 6
matrix_2[c(1, 2), 3] # full third column
#> [1] 5 6
matrix_2[, 3] # same
#> [1] 5 6
matrix_2[,-3] # remove third column
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
matrix_2[2, 3] <- NA # replace value with missing
matrix_2
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4   NA
rownames(matrix_1) <- c("a", "b") # modify row names
```

```
matrix_1
#>   [,1] [,2] [,3]
#> a    1    3    5
#> b    2    4    6
colnames(matrix_1) <- c("A", "B", "C") # modify column names
matrix_1
#>   A B C
#> a 1 3 5
#> b 2 4 6
matrix_1["a","C"] # access by row and column names
#> [1] 5
```

Data frames

Data frames are heterogeneous collctions of equal-length vectors. They are two dimensional. Use `[` or `$` to access data frames.

```
data_1 <- data.frame("A" = c(1:6),
                     "B" = rep("a", times = 6),
                     "C" = c(seq(from = 0, to = 1, by = 0.2))) # create data frame

print(data_1) # You can also use View(data_1)
#>   A B   C
#> 1 1 a 0.0
#> 2 2 a 0.2
#> 3 3 a 0.4
#> 4 4 a 0.6
#> 5 5 a 0.8
#> 6 6 a 1.0

str(data_1)
#> 'data.frame':    6 obs. of  3 variables:
#>  $ A: int  1 2 3 4 5 6
#>  $ B: chr  "a" "a" "a" "a" ...
#>  $ C: num  0 0.2 0.4 0.6 0.8 1

data_2 <- data.frame("D" = c(7:12),
                     "E" = rep("b", times = 6),
                     "F" = c(seq(from = 1, to = 2, by = 0.2)))

print(data_2)
#>    D E   F
#> 1  7 b 1.0
#> 2  8 b 1.2
#> 3  9 b 1.4
#> 4 10 b 1.6
#> 5 11 b 1.8
#> 6 12 b 2.0
```

```
# to combine data frames use cbind() and rbind()
cbind(data_1, data_2) # combine column-wise, number of rows must match
#>   A B   C  D E   F
#> 1 1 a 0.0  7 b 1.0
#> 2 2 a 0.2  8 b 1.2
#> 3 3 a 0.4  9 b 1.4
#> 4 4 a 0.6 10 b 1.6
#> 5 5 a 0.8 11 b 1.8
#> 6 6 a 1.0 12 b 2.0

rbind(data_1, data_1) # combine row_wise, column names and number of columns must
        match
#>    A B   C
#> 1  1 a 0.0
#> 2  2 a 0.2
#> 3  3 a 0.4
#> 4  4 a 0.6
#> 5  5 a 0.8
#> 6  6 a 1.0
#> 7  1 a 0.0
#> 8  2 a 0.2
#> 9  3 a 0.4
#> 10 4 a 0.6
#> 11 5 a 0.8
#> 12 6 a 1.0

# access via `[`
data_1[,"B"] # access column B
#> [1] "a" "a" "a" "a" "a" "a"
data_1[2,3] # access second row third column
#> [1] 0.2
# access via `$`
data_1$B # access column B
#> [1] "a" "a" "a" "a" "a" "a"
data_1$C[3] # access third value of column C
#> [1] 0.4
data_1[data_1$C < 0.5,] # all rows for which the values in column C are below 0.5
#>   A B   C
#> 1 1 a 0.0
#> 2 2 a 0.2
#> 3 3 a 0.4
```

Lists

Lists are heterogeneous collections of data structures. Lists are accessed with the `[` and `[[` operators.

```
list_1 <- list(1:5, c("this", "is", "the second", "vector"), matrix_1)
list_1
#> [[1]]
```

```
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "this"       "is"         "the second" "vector"
#>
#> [[3]]
#>   A B C
#> a 1 3 5
#> b 2 4 6
str(list_1) # structure of a list
#> List of 3
#>  $ : int [1:5] 1 2 3 4 5
#>  $ : chr [1:4] "this" "is" "the second" "vector"
#>  $ : int [1:2, 1:3] 1 2 3 4 5 6
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:2] "a" "b"
#>   .. ..$ : chr [1:3] "A" "B" "C"
length(list_1) # number of list elements
#> [1] 3
# to combine lists use c()
list_2 <- list(6:10, rep("a", times = 5))
list_3 <- c(list_2, list_1) # combine lists in order
list_3
#> [[1]]
#> [1]  6  7  8  9 10
#>
#> [[2]]
#> [1] "a" "a" "a" "a" "a"
#>
#> [[3]]
#> [1] 1 2 3 4 5
#>
#> [[4]]
#> [1] "this"       "is"         "the second" "vector"
#>
#> [[5]]
#>   A B C
#> a 1 3 5
#> b 2 4 6

# you can provide names to list elements as to vector elements
list_1 <- setNames(object = list_1, nm = c("a", "b", "c"))
list_1
#> $a
#> [1] 1 2 3 4 5
#>
#> $b
#> [1] "this"       "is"         "the second" "vector"
#>
#> $c
#>   A B C
```

```
#> a 1 3 5
#> b 2 4 6
# access works same as described above and below but use `[[` to select list elements
list_3[[3]] # third element in list
#> [1] 1 2 3 4 5
list_3[[5]][,"B"] # fifth element in list (a matrix) and column "B" from the matrix
#> a b
#> 3 4
list_3[1:3] # first three list elements
#> [[1]]
#> [1]  6  7  8  9 10
#>
#> [[2]]
#> [1] "a" "a" "a" "a" "a"
#>
#> [[3]]
#> [1] 1 2 3 4 5
```

Attributes

Attributes store metadata about an object.

```
attributes(results) # a named vector
#> NULL
attributes(matrix_1) # a matrix
#> $dim
#> [1] 2 3
#>
#> $dimnames
#> $dimnames[[1]]
#> [1] "a" "b"
#>
#> $dimnames[[2]]
#> [1] "A" "B" "C"
attributes(list_1) # a named list
#> $names
#> [1] "a" "b" "c"
attributes(data_1) # data frame
#> $names
#> [1] "A" "B" "C"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3 4 5 6
# or use dim(), names(), class()
```

4. R Packages

Package

Packages are similar to libraries in other programming languages. While base R is powerful, it has limited functionality and some tasks that are in principle solvable with base R can be coded more easily with specialized packages. R packages are primarily distributed via the CRAN package repository, which currently hosts more than 14,000 packages.

Install a package

To install a package from CRAN use `install.packages()` and provide a package name or a vector of package names. You need to do this only once. For instance, for the ‘dplyr’ package type `install.packages("dplyr")`, then type `library(dplyr)` to attach the dplyr package and make it available in your current R session. Using `?` to learn about a package, e.g., ?dplyr, works only if the package authors have built this feature into their package. In each session you have to load/attach the packages you want to use. It is good practice to source packages from a packages script on start up (not covered here). To check which packages are currently attached use `(.packages())`. To detach a package use `detach("package name", unload=TRUE)`.

5. Working with data

Import data sets

How you import data into R depends on the data format you are confronted with. In the following, you will deal with a .csv (comma separated values) file, which is quite common. Note that all string variables are automatically transformed to factor variables (i.e., categorical variables). This is a nuisance in R and often makes no sense. To avoid this, use the `stringsAsFactors = FALSE` argument. Following file is separated by `;` not comma, so `sep` argument is used.

```
keyword.counts <- read.csv("keyword_counts.csv",header = TRUE, sep = ";",
          stringsAsFactors = FALSE)

str(keyword.counts)
#> 'data.frame':    38 obs. of  6 variables:
#>  $ day                   : chr  "2018-10-07 00:00:00 +0200" "2018-10-08 00:00:00
        +0200" "2018-10-09 00:00:00 +0200" "2018-10-10 00:00:00 +0200" ...
#>  $ Umvolkung             : int  109 98 273 130 75 97 46 60 63 173 ...
#>  $ Großer.Austausch      : int  0 0 1 0 1 0 0 0 0 0 ...
#>  $ Bevölkerungsaustausch: int  10 27 14 72 112 102 15 50 42 39 ...
#>  $ CDU                   : int  0 3 1 2 2 4 2 1 0 1 ...
#>  $ AfD                   : int  14 3 5 24 18 10 19 17 38 30 ...
head(keyword.counts)
#>                           day Umvolkung Großer.Austausch Bevölkerungsaustausch
#> 1 2018-10-07 00:00:00 +0200       109                0                    10
```

```
#> 2 2018-10-08 00:00:00 +0200        98           0                    27
#> 3 2018-10-09 00:00:00 +0200       273           1                    14
#> 4 2018-10-10 00:00:00 +0200       130           0                    72
#> 5 2018-10-11 00:00:00 +0200        75           1                   112
#> 6 2018-10-12 00:00:00 +0200        97           0                   102
#>   CDU AfD
#> 1   0  14
#> 2   3   3
#> 3   1   5
#> 4   2  24
#> 5   2  18
#> 6   4  10
tail(keyword.counts)
#>                           day Umvolkung Großer.Austausch Bevölkerungsaustausch
#> 33 2018-11-07 23:00:00 +0100        99                0                    49
#> 34 2018-11-08 23:00:00 +0100        77                1                     4
#> 35 2018-11-09 23:00:00 +0100       107                0                     6
#> 36 2018-11-10 23:00:00 +0100        92                3                     4
#> 37 2018-11-11 23:00:00 +0100        80                1                     8
#> 38 2018-11-12 23:00:00 +0100        52                0                     1
#>    CDU AfD
#> 33  11  27
#> 34  89  16
#> 35  43  14
#> 36  10  49
#> 37  41  30
#> 38   3  19
dim(keyword.counts)
#> [1] 38  6
```

Note that `read.csv` can be very slow for huge dataset. In such cases I recommend `fread()` from the 'data.table' package or even way faster `vroom` from the 'vroom' package. For .txt files that store text, use base R's readLines(), functions from the 'readtext' package, etc., really depends on where you want to go. For SPSS or STATA files try the 'haven' package.

Apply family of functions

The `apply()` family pertains to the R base package and is populated with functions to manipulate slices of data from matrices, arrays, lists and dataframes in a repetitive way. These functions allow crossing the data in a number of ways and avoid explicit use of loop constructs. They act on an input list, matrix or array and apply a named function with one or several optional arguments.

To apply functions on matrices and arrays, the structure of the function call is `apply(data, rows or columns (margin), function to apply)`.

```
# Create a new object storing sum of three variables
umvolkung_sum <- apply(keyword.counts[c(2,3,4)], 1, sum) #when MARGIN=1, it applies
        over rows, whereas with MARGIN=2, it works over columns.
```

For more about `apply()` family, check `lapply()`, `sapply()`, `tapply()`.

Data management

For data management purposes, the 'dplyr' package provides a handy grammar for data manipulation. You can do almost all of this with base R, dplyr is just much more convenient, especially when combined with the pipe (not covered here).

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- mutate() adds new variables that are functions of existing variables
- select() picks variables based on their names.
- filter() picks cases based on their values.
- summarise() reduces multiple values down to a single summary.
- arrange() changes the ordering of the rows. These all combine naturally with group_by() which allows you to perform any operation "by group".

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

To change variable names, use `rename()`.

```
# To change name A to B, write B = A.
keyword.counts <- rename(keyword.counts,
                         A = Umvolkung,
                         B = Großer.Austausch,
                         C = Bevölkerungsaustausch)
```

To select or reorder columns conditional on specific criteria, use `select()`.

```
names(keyword.counts) # show column names
#> [1] "day" "A"   "B"   "C"   "CDU" "AfD"

three_keywords <- select(keyword.counts, A, B, C) # select variables and reorder by
        name

three_keywords2 <- select(keyword.counts, c(2:4)) # select variables and reorder by
        position
```

To add new or alter existing variables, use `mutate()`. This example is also using pipes, %>%. Pipes take the output from one function and feed it to the first argument of the next function.

```r
keyword.counts <- keyword.counts %>% # This is a pipe!
  mutate(
    Total = A + B + C
    )

# Above code is the same as below:
keyword.counts <- mutate(keyword.counts,
                         Total = A + B + C)

# 1. Change day variable from string to "Date" class object and 2. create "months"
        variable
keyword.counts <- keyword.counts %>%
  mutate(
    day = as.Date(day),
    month = months(day)
  )
```

For detail of `as.Date` function, see here.

To select rows conditional on specific criteria, use `filter()`.

```r
# Get rows whose Total number exceeds 150.
keyword.counts %>%
  filter(Total > 150)
#>            day   A  B   C CDU AfD Total    month
#> 1  2018-10-09 273  1  14   1   5   288  October
#> 2  2018-10-10 130  0  72   2  24   202  October
#> 3  2018-10-11  75  1 112   2  18   188  October
#> 4  2018-10-12  97  0 102   4  10   199  October
#> 5  2018-10-16 173  0  39   1  30   212  October
#> 6  2018-10-28 140  0  12   4  13   152  October
#> 7  2018-11-01 197  7  15   1 106   219 November
#> 8  2018-11-02 233 14  30  23 147   277 November
#> 9  2018-11-03 230  6  29   5  57   265 November
#> 10 2018-11-04 348  0   4   7 279   352 November
#> 11 2018-11-05 227  0  21  11 161   248 November
#> 12 2018-11-06 133  1  32  11  78   166 November
```

To order rows by variables, use `arrange()`.

```r
keyword.counts %>%
  filter(Total > 150) %>%
  arrange(desc(Total)) # sort a variable in descending order.
#>            day   A  B   C CDU AfD Total    month
#> 1  2018-11-04 348  0   4   7 279   352 November
#> 2  2018-10-09 273  1  14   1   5   288  October
#> 3  2018-11-02 233 14  30  23 147   277 November
#> 4  2018-11-03 230  6  29   5  57   265 November
#> 5  2018-11-05 227  0  21  11 161   248 November
#> 6  2018-11-01 197  7  15   1 106   219 November
```

```
#> 7  2018-10-16 173  0  39   1  30   212  October
#> 8  2018-10-10 130  0  72   2  24   202  October
#> 9  2018-10-12  97  0 102   4  10   199  October
#> 10 2018-10-11  75  1 112   2  18   188  October
#> 11 2018-11-06 133  1  32  11  78   166 November
#> 12 2018-10-28 140  0  12   4  13   152  October
```

summarise() creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. To group data by one or more variables in order to perform group-specific operations, use groub_by.

```
keyword.counts %>%
  summarise(
    n = n(),
    A_sum = sum(A),
    A_mean = mean(A))
#>    n A_sum   A_mean
#> 1 38  4227 111.2368

# Using group_by
keyword.counts %>%
  group_by(month) %>%
  summarise(
    n = n(),
    Total_sum = sum(Total)
  )
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 2 x 3
#>   month        n Total_sum
#>   <chr>    <int>     <int>
#> 1 November    12      2111
#> 2 October     26      3106
```

6. Further topics and ressources

How to write a good code

- use Comments otherwise you forget.
- use ‘#’ sign to indicate comments. R ignore the line start with the sign.
- write code with a consistent style.
- For example: Google’s style guide

Where to go next

- improving code readability with the pipe operator (‘magrittr’ package)

- improving coding and documentation practice with R Markdown
- managing your file system
- working with relational data and databases
- working with strings and dates
- building functions
- mastering graphics
- discovering textual, spatial, and network data
- discovering distributions and statistical models
- automating Web data extraction
- optimizing your code via vectorization and data.table
- learning about packages that make it easier to work with R

Where to look

Books

- Imai, Kosuke. 2017. Quantitative social science. An introduction. Princeton, NJ: Princeton University Press.
- Munzert, Simon, Christian Rubba, Peter Meißner, Dominic Nyhuis. 2015. Automated data collection with R. A practical guide to Web scraping and text mining. Chichester: Wiley.
- Wickham, Hadley. 2014. Advanced R. Boca Raton: CRC Press.
- Wickham, Hadley. 2009. Ggplot2. Elegant graphics for data analysis. New York: Springer
- Wickham, Hadley and Garrett Grolemund. R for data science. Sebastopol, CA: O'Reilly.

Online ressources

- https://stackoverflow.com/
- https://www.r-bloggers.com/
- https://cran.r-project.org/web/views/
- https://journal.r-project.org/
- https://www.rstudio.com/resources/cheatsheets/
- http://style.tidyverse.org/
- http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html
- http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Recommended packages

- stringr - provides common string operations
- pacman - manage package installation and sourcing
- plyr - split-apply-combine paradigm
- dplyr - successor of plyr, tailored for data frames
- data.table - enhanced (fast and memory efficient) data.frame
- haven - import and export 'SPSS', 'Stata' and 'SAS' Files
- magrittr - provides the pipe operator
- ggplot2 - data visualization using the grammar of graphics, base R graphics can do just fine, though
- survey - analysis of complex survey samples
- writexl - read, write, and edit XLSX Files
- lubridate - dealing with dates
- zoo - dealing with time series
- eeptools - misc convenience functions
- httr - tools for working with URLs and HTTP

- rvest - tools for Web scraping
- XML - tools for parsing and generating XML
- crayon - colored terminal output