Live Coding Sept. 13

R-team

9/13/2019

Session 1: defaults in functions

```
Multiply <- function(x, y = x){
  x * y
}
```

```
Multiply(2, 4)
```

```
## [1] 8
```

```
Multiply(2)
```

```
## [1] 4
```

Session 2: Creating a matrix

```
my_matrix <- matrix(1:24, ncol=6, byrow=FALSE)
```

Note, the `matrix()` function automatically infers the number of rows that are needed.

```
my_matrix[1, 2]
```

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

Some useful operations on / for matrices:

```
colnames(my_matrix) <- letters[1:6]
rownames(my_matrix) <- LETTERS[1:4]
```

Remember indexing with the tags/ labels / names?

```
my_matrix["A", ]
```

```
##  a  b  c  d  e  f
##  1  5  9 13 17 21
```

```
my_matrix[, "a"]
```

```
## A B C D
## 1 2 3 4
```

```
my_matrix["A", , drop=FALSE]
```

```
##   a b c  d  e  f
## A 1 5 9 13 17 21
```

```
my_matrix[, "a", drop=FALSE]
```

```
##   a
## A 1
## B 2
## C 3
## D 4
```

Do you understand the point of using `drop = FALSE`? vector vs matrix (using class()to tell)

```
my_second_matrix <- 1:24
names(my_second_matrix) <- letters[1:24]
# another way to create a matrix
dim(my_second_matrix) <- c(4, 6)
my_second_matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    5    9   13   17   21
## [2,]    2    6   10   14   18   22
## [3,]    3    7   11   15   19   23
## [4,]    4    8   12   16   20   24
```

```
my_second_matrix["f"]
```

```
## [1] NA
```

```
names(my_second_matrix) <- letters[1:24]
```

array:

```
my_array <- array(1:64, dim = c(4, 4, 4))
my_array
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
##
## , , 2
##
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   17   21   25   29
## [2,]   18   22   26   30
## [3,]   19   23   27   31
## [4,]   20   24   28   32
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   33   37   41   45
## [2,]   34   38   42   46
## [3,]   35   39   43   47
## [4,]   36   40   44   48
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,]   49   53   57   61
## [2,]   50   54   58   62
## [3,]   51   55   59   63
## [4,]   52   56   60   64
```

```
colSums(my_array)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   74  138  202
## [2,]   26   90  154  218
## [3,]   42  106  170  234
## [4,]   58  122  186  250
```

Session 3: accessing it's elements

First create some 100 draws from standard normal distribution:

```
my_vector <- rnorm(100)
```

(remember: look at `?rnorm` for more details)

```
variance <- var(my_vector)
```

Let's make a list, and remember we can use tags

```
my_data_summary <- list(
  mean = mean(my_vector),
  median = median(my_vector),
  sd(my_vector),
  variance,
  range = range(my_vector),
  min_max = c(
    min = min(my_vector),
    max = max(my_vector)
```

```
  )
)
my_data_summary
```

```
## $mean
## [1] -0.08902809
##
## $median
## [1] -0.04906347
##
## [[3]]
## [1] 0.9086601
##
## [[4]]
## [1] 0.8256632
##
## $range
## [1] -2.539605  1.904231
##
## $min_max
##       min       max
## -2.539605  1.904231
```

Thus `my_data_summary` is a list, which is a vector of list elements. A list element is a container that can take any class and mode.

Accessing an element inside a list:

```
my_data_summary[[1]]
```

Number, name or using $

```
## [1] -0.08902809
```

```
my_data_summary[["mean"]]
```

```
## [1] -0.08902809
```

```
my_data_summary$median
```

```
## [1] -0.04906347
```

How about?

```
my_data_summary$med
```

只使用前面几个字母

```
## [1] -0.04906347
```

```
my_data_summary$me
```

```
## NULL
```

Accessing an element in a vector **inside** a list:

```r
my_data_summary$min_max[1]
```

```
##       min
## -2.539605
```

```r
my_data_summary[["min_max"]][2]
```

```
##      max
## 1.904231
```

```r
my_data_summary[["min_max"]]["max"]
```

```
##      max
## 1.904231
```

Wrong way of accessing elements in a list / vector:

```r
my_data_summary["min_max"]
```

```
## $min_max
##       min       max
## -2.539605  1.904231
```

```r
my_data_summary["min_max"][2]
```

```
## $<NA>
## NULL
```

```r
my_data_summary[["min_max"]]$max
```

```
## Error in my_data_summary[["min_max"]]$max: $ operator is invalid for atomic vectors
```

```r
my_list[["mat"]][1, 1]
```

```
## Error in eval(expr, envir, enclos): object 'my_list' not found
```

```r
my_list["mat"][1, 1]
```

```
## Error in eval(expr, envir, enclos): object 'my_list' not found
```

```r
my_list["mat"][1][1][1][1][1]
```

```
## Error in eval(expr, envir, enclos): object 'my_list' not found
```

About `unlist()`

We can also create an empty list with elements

```
vector("list", 10)
```

mode: default is logical
similar: logical(10)

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

You can 'unlist' a list, if components are simple enough:

```
my_list <- rep(list(0), 10)
unlist(my_list)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

and even recursively

```
my_list <- rep(list(list(c(0, 1))), 10)
unlist(my_list, recursive=TRUE)
```

递归, e.g. a list inside a list

```
##  [1] 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

```
my_list <- rep(list(diag(1:3)), 10)
unlist(my_list)
```

```
##  [1] 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0
## [36] 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0
## [71] 0 3 1 0 0 0 2 0 0 0 3 1 0 0 0 2 0 0 0 3
```

```r
my_list <- rep(list(function(x){return(x)}), 10)
unlist(my_list)
```

```
## [[1]]
## function(x){return(x)}
##
## [[2]]
## function(x){return(x)}
##
## [[3]]
## function(x){return(x)}
##
## [[4]]
## function(x){return(x)}
##
## [[5]]
## function(x){return(x)}
##
## [[6]]
## function(x){return(x)}
##
## [[7]]
## function(x){return(x)}
##
## [[8]]
## function(x){return(x)}
##
## [[9]]
## function(x){return(x)}
##
## [[10]]
## function(x){return(x)}
```

Session 4: names / tags on data.frame objects

Like with lists, you can create 'tags'

```r
heart_rate <- sample(50:90, 20, replace = T)
person_id <- 1:20
my_data <- data.frame(
  id = person_id,
  heart_rate,
  rnorm(20, mean=100, sd=15)
)
```

Different ways of accessing a data.frame:

```r
head(my_data, 5)
```

```
##   id heart_rate rnorm.20..mean...100..sd...15.
## 1  1         78                       84.19907
```

```
## 2  2         58                    91.67257
## 3  3         69                   123.34904
## 4  4         57                   106.21811
## 5  5         59                    95.43636
```

`list`-like:

```
my_data[[1]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
my_data[["id"]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
my_data[1] # is special! why?
```

stay as dataframe

```
##    id
## 1   1
## 2   2
## 3   3
## 4   4
## 5   5
## 6   6
## 7   7
## 8   8
## 9   9
## 10 10
## 11 11
## 12 12
## 13 13
## 14 14
## 15 15
## 16 16
## 17 17
## 18 18
## 19 19
## 20 20
```

`matrix`-like:

```
my_data[1, 2]
```

```
## [1] 78
```

Properties of a data.frame..

```
nrow(my_data)
```

```
## [1] 20
```

```
ncol(my_data)
```

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

```
dim(my_data)
```

```
## [1] 20  3
```

... If there is time left... some real live coding about `attributes`?

Session 5: getting a file

```
getwd()
```

```
## [1] "/Users/maartenkampert/Dropbox/0_Teaching/SCR/R_1920/Week02_Basics2"
```

```
# setwd()
```

With regards to a `.txt` file: actually it is not the file type (`.txt`) that is important, but the way the information is encoded. File type extensions such as `.txt` or `.csv` do not actually mean anything other than to tell your PC what type of behaviour to choose when it comes to opening files: e.g. a `.txt` file is opened with Notepad (Microsoft) or TextEdit (MacOs) and `.csv` is usually opened with e.g. Excel or similar software.

NB. what may happen is: a comma seperated file, saved as a `.txt` file will give you strange results in R!

Here is an example:

```
my_data_frame <- data.frame(
  var1 = rnorm(20),
  var2 = rnorm(20)
)
```

```
?write.table
```

Do you see in the helpfile how only 2 arguments have no default?

Let's save the data

```
write.table(my_data_frame, file="0_data/my_data.csv")
```

Can we find our created file? Remember `swirl`, we can use a "relative" path to the file:

```
list.files() # remember swirl
# rstudio files
read.csv("0_data/my_data.csv")
read.table("0_data/my_data.csv")
```

Or absolute:

```
read.table("/Users/maartenkampert/Dropbox/0_Teaching/SCR/R_1920/Week02_Basics2/0_data/my_data.csv")
```

```
##           var1         var2
## 1  -1.16120982 -1.472672252
## 2   0.28021410 -0.646576462
## 3   1.03402216 -0.203880461
## 4   1.30128991  1.685803251
## 5  -0.45532461 -0.239373267
## 6   0.59869904 -0.744524953
## 7  -0.47549623 -0.606963251
## 8   1.11489670  0.001023536
## 9  -1.77343375 -0.803317659
## 10  1.32582618 -1.243519922
## 11 -0.08097418 -1.422151078
## 12 -1.42456220  0.406853100
## 13  0.11399908 -0.692657307
## 14  1.62500484  0.160419097
## 15 -0.17591907 -0.416807092
## 16 -0.10463843  0.018919862
## 17 -0.28806233  0.009298031
## 18  1.81957330  0.226939437
## 19  0.88872165  2.581401209
## 20  1.79104326  2.305143463
```

The directory adress is too long! You cannot even read it on the 'knitted' `.pdf` file of this live-coding session...

For writing files, similar code has to be used (check-out the `swirl` module or Matloff?)

Session 6: `apply()`

```
my_vector <- 1:5
sum(my_vector)
```

```
## [1] 15
```

```
my_matrix
```

```
##   a b  c  d  e  f
## A 1 5  9 13 17 21
## B 2 6 10 14 18 22
## C 3 7 11 15 19 23
## D 4 8 12 16 20 24
```

```
rowSums(my_matrix)
```

```
##  A  B  C  D
## 66 72 78 84
```

```r
1+5+9+13+17+21
```

```
## [1] 66
```

```r
colSums(my_matrix)
```

```
##  a  b  c  d  e  f
## 10 26 42 58 74 90
```

```r
1 + 2 + 3 + 4
```

```
## [1] 10
```

```r
prod(my_vector)
```

```
## [1] 120
```

```r
rowProd(my_matrix)
```

```
## Error in rowProd(my_matrix): could not find function "rowProd"
```

hmz....

Try the non anonymous(!) version in apply:

```r
apply(my_matrix, 1, prod)
```

```
##       A       B       C       D
##  208845  665280 1514205 2949120
```

```r
1*5*9*13*17*21
```

```
## [1] 208845
```

```r
apply(my_matrix, 2, prod)
```

```
##      a      b      c      d      e      f
##     24   1680  11880  43680 116280 255024
```

```r
1*2*3*4
```

```
## [1] 24
```

or the anonymous version:

```r
apply(my_matrix, 1, function(x){
  x[1] + tail(x, 1)
})
```

```
##  A  B  C  D
## 22 24 26 28
```

```r
# 1+21; 2+22; 3+23; 4+24
```

Take care of the output of the function you give to apply! R will try to fit all of the returned values into a neat format. Try for example:

```r
apply(my_matrix, 2, function(x){
  list(mean = mean(x), sd = sd(x))
})
```

```
## $a
## $a$mean
## [1] 2.5
##
## $a$sd
## [1] 1.290994
##
##
## $b
## $b$mean
## [1] 6.5
##
## $b$sd
## [1] 1.290994
##
##
## $c
## $c$mean
## [1] 10.5
##
## $c$sd
## [1] 1.290994
##
##
## $d
## $d$mean
## [1] 14.5
##
## $d$sd
## [1] 1.290994
##
##
## $e
## $e$mean
## [1] 18.5
##
```

```
## $e$sd
## [1] 1.290994
##
##
## $f
## $f$mean
## [1] 22.5
##
## $f$sd
## [1] 1.290994
```

or

```
apply(my_matrix, 2, function(x){
  matrix(c(mean(x), sd(x)), ncol=1)
})
```

```
##             a        b         c         d         e         f
## [1,] 2.500000 6.500000 10.500000 14.500000 18.500000 22.500000
## [2,] 1.290994 1.290994  1.290994  1.290994  1.290994  1.290994
```

There are other apply functions, these will be discussed in later lectures. One of these functions, `lapply()` will already appear in the (self-study) exercises.