SCR week 3: exercises

Exercises: part 1

1.1 Elipses…

Before you start this exercise, make sure you have done the `swirl` module 9 on functions, to enhance your understanding of `list(...)` inside a function.

The elipses, `...`, are special. You can look at the arguments passed through elipses using `list(...)`. As with any other list, it is possible to access it's contents, to read out the value of the arguments. The names of the list are equal to the argument names. It is useful for passing on many arguments at the same time, possible arguments you do not necessarily always use.

a.

Pass some arguments through the following functions to see what the return value is.

```
MyFlexSummary <- function(...){
  return(list(...))
}
```

b.

Replace `list` in the last function, with the `mean` function. Call `MyFlexSummary`. Does it work? You should receive an error. See if you can provide some arguments that allows you to call the function without an error and such that it also returns a mean (of some vector you provide).

Answer:

```
MyFlexSummary <- function(...){
  return(mean(...))
}

# MyFlexSummary() # does not work
MyFlexSummary(x = 1:100)
```

```
## [1] 50.5
```

c.

Add a few lines of code such that not just the mean, but also the `sd` of a vector `x` is calculated. Return the results of both as a list. Give some nice names the elements of the list you return. Make sure the argument of `MyFlexSummary` stays just `...`.

Answer:

```r
MyFlexSummary <- function(...){

  my_mean <- mean(...)
  my_sd <- sd(...)

  return_list = list(
    mean = my_mean,
    sd = my_sd
  )

  return(return_list)
}

MyFlexSummary(1:100)
```

```
## $mean
## [1] 50.5
##
## $sd
## [1] 29.01149
```

d.

Call `MyFlexSummary` also with a `trim` argument (as you would use in `mean`). Does this work?

e.

You should have encountered an error. We will fix this by making a 'wrapper'. In this case a custom 'shell' around the function `sd`, so that it performs, preferably, in the exact same way if used like we would `sd` but can do more, if we choose to. Create a function called `GiveSD`, that takes as arguments: `x`, `na.rm` and `...`. Return the standard deviation of `x`, using a call to `sd` with the `x` and `na.rm` arguments. Like in `sd`, set a default for `na.rm`. Do nothing with the arguments in `...`.

```r
GiveSD <- function(x, na.rm = FALSE, ...){
  return(sd(x, na.rm))
}
```

f.

Inside `MyFlexSummary` replace `sd` with the 'wrapper' `GiveSD()`. Call it, also providing an argument for `trim`. Does it work now?

Answer:

```r
MyFlexSummary <- function(...){

  my_mean <- mean(...)
  my_sd <- GiveSD(...)

  return_list = list(
    mean = my_mean,
    sd = my_sd
  )
  names(return_list) <- c("mean", "sd")

  return(return_list)
}

MyFlexSummary(x = rnorm(100), trim = 0.1)
```

```
## $mean
## [1] 0.03972346
##
## $sd
## [1] 0.9304026
```

g.

Now call the function `MyFlexSummary` giving a vector with an `NA`. Is it possible to override the default argument for na.rm, by specifying it inside the elipsis `...`?

Answer:

Yes.

```r
MyFlexSummary(c(rnorm(100), NA), na.rm = TRUE, trim = 0.1)
```

```
## $mean
## [1] 0.04342624
##
## $sd
## [1] 0.9204143
```

Outro

After we've seen the programming structures part about `if-else` statements, we can do some more interesting stuff, especially with regards to wrappers. Suppose you'd not just want to calculate a trimmed mean, but also a standard deviation using the trimmed mean. Using the `MySd` wrapper you could redefine the behaviour of `sd` based on whether the `trim` argument was provided or not.

Exercises part 2

2.1 Working with a table

For the following exercises, take the code below to create a `data.frame` with some data.

```
set.seed(2017)
N <- 500
my_data_frame <- data.frame(
  species = sample(
    c("elephant", "giraffe", "monkey", "snake"),
    N,
    replace = TRUE
  ),
  hair_colour = sample(
    c("blonde", "brown", "red"), N, replace = TRUE
  ),
  iq = sample(
    c("70-79", "80-89", "90-99", "100-109", "110-119", "120-129"),
    N,
    replace = T
  )
)
```

a.

Look at `my_data_frame$iq`. How is the ordering of the levels of this factor? Use the code below, and then look at the `iq` again. Did the data change? How about the ordering?

```
my_data_frame$iq <- factor(my_data_frame$iq, levels = levels(factor(my_data_frame$iq))[c(4, 5, 6, 1, 2,
```

Answer:

The levels of the factor are ordered according to alphabet of the values of the factors (e.g. 100 comes before 70 alphabetically). Using the code we changed the levels, but not the data.

b.

Create a cross-table of the entire data set. Possibly it is convenient to switch around the positions of the columns, before you create a table.

Answer: I find it easiest to have as few 3rd dimension 'layers' as possible, in this case, haircolour.

```
my_table <- table(my_data_frame[, c(1, 3, 2)])
```

交换位置

另外可以用order(..$..，来排序)

c.

How many blonde monkey's, with an IQ score between 110 and 119 are there?

Answer:

```r
# read out from:
my_table
```

```
## , , hair_colour = blonde
## 
##           iq
## species    70-79 80-89 90-99 100-109 110-119 120-129
##   elephant    13     9     6       4       4       9
##   giraffe      6     9     5       6       6       7
##   monkey       2     8     7       2       9      13
##   snake        6     5     7       4      10       5
## 
## , , hair_colour = brown
## 
##           iq
## species    70-79 80-89 90-99 100-109 110-119 120-129
##   elephant     3     3     2      12       7      12
##   giraffe      5     9    13       8       8       5
##   monkey       5     4     2      14       9       6
##   snake        5     9     9       6       8       7
## 
## , , hair_colour = red
## 
##           iq
## species    70-79 80-89 90-99 100-109 110-119 120-129
##   elephant     8     8     5      11       8      12
##   giraffe      3     2     5       3       7       4
##   monkey       7     8    13       7       4       8
##   snake        8    11     8       5       8       4
```

```r
# or:
my_table["monkey", "110-119", "blonde"]
```

```
## [1] 9
```

d.

How many animals have red haircolour?

Answer: e.g.:

```r
apply(my_table, 3, sum)["red"]
```

```
## red
## 167
```

```r
sum(my_table[, , "red"])
```

```
## [1] 167
```

e.

Of the animals with an IQ score between 90 and 99, how many animals are there of each species?

Answer: e.g.

```r
rowSums(my_table[, 3,])
```

```
## elephant  giraffe   monkey    snake 
##       13       23       22       24
```

f.

Extract from your 3D table, a table containing the number of occurences of animals with particular haircolour, *regardless* of their IQ score. Try to use the apply function.

Answer:

```r
apply(my_table, c(1, 3), sum)
```

```
##           hair_colour
## species    blonde brown red
##   elephant     45    39  52
##   giraffe      39    48  24
##   monkey       41    40  47
##   snake        37    44  44
```

```r
# or cheating:
table(my_data_frame[, -3])
```

```
##           hair_colour
## species    blonde brown red
##   elephant     45    39  52
##   giraffe      39    48  24
##   monkey       41    40  47
##   snake        37    44  44
```