R Scripting

Exercises for unit 3 - Structured programming

Marcus Wurzer

Please solve the following problems!

1. Coursebook exercises:

   a. Use vapply() to:

      - Compute the standard deviation of every column in a numeric data frame (Hint: Type `help(package = "datasets")` and choose a suitable data set).
      - Compute the standard deviation of every numeric column in a mixed data frame. (Hint: you'll need to use `vapply()` twice.)

      As a numeric data.frame we choose `cars`:

      ```
      vapply(cars, sd, numeric(1))
      ```

      ```
      ##     speed      dist
      ##  5.287644 25.769377
      ```

      And as a mixed data.frame we choose `iris`:

      ```
      vapply(iris[vapply(iris, is.numeric, logical(1))], sd, numeric(1))
      ```

      ```
      ## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
      ##    0.8280661    0.4358663    1.7652982    0.7622377
      ```

   b. What does `replicate()` do? What sort of for loop does it eliminate? Why do its arguments differ from `lapply()` and friends?

      As stated in `?replicate`:

      `replicate` is a wrapper for the common use of `sapply` for repeated evaluation of an expression (which will usually involve random number generation).

      Like `sapply()` `replicate()` eliminates a for loop. As explained for `Map()` in the textbook, also every `replicate()` could have been written via `lapply()`. But using `replicate()` is more concise, and more clearly indicates what you're trying to do.

   c. What's the relationship between `which()` and `Position()`?

      `which()` returns all indices of true entries from a logical vector. `Position()` returns just the first (default) or the last integer index of all true entries that occur by applying a predicate function on a vector. So the default relation is `Position(f, x) <=> min(which(f(x)))`.

2. Simulate some sales figures data of a shop using the following lines of code:

   ```
   set.seed(1)
   sf <- matrix(round(rnorm(144, 500, 200)), ncol = 12)
   rownames(sf) <- 2011:2022
   colnames(sf) <- abbreviate(month.name, 3, named = FALSE)
   ```

a. Use a functional to compute the average monthly sales figures over the whole period!

```
apply(sf, 2, mean)
```

```
##      Jnr      Fbr      Mrc      Apr      May      Jun      Jly      Ags
## 553.7500 506.2500 478.0000 533.0000 536.8333 571.3333 445.0000 616.5833
##      Spt      Oct      Nvm      Dcm
## 464.2500 514.5000 490.1667 389.5833
```

```
# or
colMeans(sf)
```

```
##      Jnr      Fbr      Mrc      Apr      May      Jun      Jly      Ags
## 553.7500 506.2500 478.0000 533.0000 536.8333 571.3333 445.0000 616.5833
##      Spt      Oct      Nvm      Dcm
## 464.2500 514.5000 490.1667 389.5833
```

b. Now generate a data frame to have the data represented in a different way:

```
sf.df <- data.frame(sales = c(t(sf)),
                    month = factor(colnames(sf), levels = colnames(sf)))
```

Apply another functional on `sf.df` to get the same average monthly sales figures as above!

```
tapply(sf.df$sales, sf.df$month, mean)
```

```
##      Jnr      Fbr      Mrc      Apr      May      Jun      Jly      Ags
## 553.7500 506.2500 478.0000 533.0000 536.8333 571.3333 445.0000 616.5833
##      Spt      Oct      Nvm      Dcm
## 464.2500 514.5000 490.1667 389.5833
```

```
# or
with(sf.df, tapply(sales, month, mean))
```

```
##      Jnr      Fbr      Mrc      Apr      May      Jun      Jly      Ags
## 553.7500 506.2500 478.0000 533.0000 536.8333 571.3333 445.0000 616.5833
##      Spt      Oct      Nvm      Dcm
## 464.2500 514.5000 490.1667 389.5833
```

3. The Fibonacci sequence is a famous sequence in mathematics. The first two elements are defined as [1, 1]. Subsequent elements are defined as the sum of the preceding two elements. For example, the third element is 2 (= 1+1), the fourth element is 3 (= 1+2), the fifth element is 5 (= 2+3), and so on.

   To obtain the first 12 Fibonacci numbers in R, we can use

```
Fibonacci <- numeric(12)
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:12) Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
```

   Modify the code to generate the Fibonacci sequence in the following ways:

   a. Change the first two elements to 2 and 2.
   b. Change the first two elements to 3 and 2.
   c. Change the update rule from summing successive elements to taking differences of successive elements. For example, the third element is defined as the second element minus the first element, and so on.
   d. Change the update rule so that each element is defined as the sum of three preceding elements. Set the third element as 1 in order to start the process.

```r
Fibonacci <- numeric(12)
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:12) Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]

# a.
Fibonacci[1] <- Fibonacci[2] <- 2
for (i in 3:12) Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
Fibonacci
```

```
##  [1]   2   2   4   6  10  16  26  42  68 110 178 288
```

```r
# b.
Fibonacci[1] <- 3
Fibonacci[2] <- 2
for (i in 3:12) Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
Fibonacci
```

```
##  [1]   3   2   5   7  12  19  31  50  81 131 212 343
```

```r
# c.
for (i in 3:12) Fibonacci[i] <- Fibonacci[i - 1] - Fibonacci[i - 2]
Fibonacci
```

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

```r
# d.
Fibonacci[3] <- 1
for (i in 4:12) Fibonacci[i] <- Fibonacci[i - 3] + Fibonacci[i - 2] + Fibonacci[i - 1]
Fibonacci
```

```
##  [1]   3   2   1   6   9  16  31  56 103 190 349 642
```

4. A digitized black and white image can be represented by a binary matrix, with "0" symbolizing color value "white" and "1" color value "black".

   a. Create an 800x600 matrix and randomly fill it with black and white values.
   b. Compute the grey value (= mean color value) for each row and each column of the matrix, respectively, using a loop.
   c. Compute the grey value (= mean color value) for each row and each column of the matrix, respectively, using a functional.

```r
# a.
X <- matrix(sample(0:1, 800 * 600, replace = TRUE), nrow = 800)
dim(X)
```

```
## [1] 800 600
```

```r
# b.
res.row <- vector()
for (i in 1:nrow(X)) res.row[i] <- mean(X[i, ])
head(res.row) # Note: "head" because of the lengthy output
```

```
## [1] 0.4866667 0.5100000 0.4700000 0.4883333 0.4683333 0.5050000
```

```r
res.col <- vector()
for (i in 1:nrow(X)) res.col[i] <- mean(X[i, ])
head(res.col)
```

```
## [1] 0.4866667 0.5100000 0.4700000 0.4883333 0.4683333 0.5050000
```

```r
# c.
head(apply(X, 1, mean))
```

```
## [1] 0.4866667 0.5100000 0.4700000 0.4883333 0.4683333 0.5050000
```

```r
head(apply(X, 2, mean))
```

```
## [1] 0.50125 0.49125 0.51125 0.50875 0.50375 0.54125
```

```r
# faster equivalent to apply -> see ?rowMeans (and ?rowSums, too))
head(rowMeans(X))
```

```
## [1] 0.4866667 0.5100000 0.4700000 0.4883333 0.4683333 0.5050000
```

```r
head(colMeans(X))
```

```
## [1] 0.50125 0.49125 0.51125 0.50875 0.50375 0.54125
```

5. Data set `mtcars` is included in R:

```r
data(mtcars)
?mtcars
```

   a. Coerce variable `vs` to a factor with meaningful labels, then use a functional to return a summary of variable `disp` (displacement) - separately for cars with V-shaped and straight engines and given in ccm (cubic centimeters) instead of cubic inches (conversion formula: ccm = cu.in / 0.061024).
   b. Coerce variable `am` to a factor with meaningful labels, too, then use a functional to return the mean fuel consumption (variable `mpg`) - separately for all four combinations of motor and engine types and given in liters/km instead of miles per gallon (conversion formula: liters/km = 378.5411784 / (1.609344 * mpg)).

```r
# a.
dat <- mtcars
dat$vs <- factor(dat$vs, labels = c("V-shaped", "straight"))

tapply(dat$disp / 0.061024, dat$vs, summary)
```

```
## $`V-shaped`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1971    4520    5096    5033    5899    7735
##
## $straight
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1165    1361    1975    2171    2661    4228
```

```r
# or
attach(dat)
tapply(disp / 0.061024, vs, summary)
```

```
## $`V-shaped`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1971    4520    5096    5033    5899    7735
##
## $straight
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1165    1361    1975    2171    2661    4228
```

```r
detach(dat)
#or
with(dat, tapply(disp / 0.061024, vs, summary))
```

```
## $`V-shaped`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1971    4520    5096    5033    5899    7735
##
## $straight
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1165    1361    1975    2171    2661    4228
```

```
# b.
dat$am <- factor(dat$am, labels = c("automatic", "manual"))

with(dat, tapply(378.5411784 / (1.609344 * mpg), list(vs, am), mean))
```

```
##          automatic    manual
## V-shaped  16.18838 12.325984
## straight  11.47832  8.513789
```

6. Generate a 31x10 matrix called `temps`, containing temperature measurements (in °C) for the 31 daily average high temperatures of July in ten cities. The criteria are:

   - In each city, the temperatures should follow a normal distribution with a randomly generated mean temperature that lies between 20 and 30 degrees and a standard deviation of 5.
   - The row names should be created using the following scheme: July 1, ..., July 2, ..., July 3, ...
   - The column names should be created using the following scheme: City A, ..., City B, ..., City C, ...

```
set.seed(1)
temps <- matrix(replicate(10, round(rnorm(31, sample(20:30, 1), 5), 1)), nrow = 31)
colnames(temps) <- paste("City", LETTERS[1:10])
rownames(temps) <- paste("July", 1:31)
```

Unfortunately, many of the entries have been lost when reading the data...:

```
temps[sample(1:length(temps), length(temps) / 10)] <- NA
```

   a. Coerce the matrix to a data frame. Then, use a functional to coerce this data frame to some suitable data structure that only contains the non-missing values for each city and the respective names (= dates). (Note: Use the functional that retains the names, i.e., it is not necessary to make use of the `names()` function!)
   b. Once this new data structure has been created, operate over it to identify the days that showed the highest July temperature for each city. These days should be presented using the simplest possible data structure.

```
# a.
temps <- data.frame(temps)
temps.l <- apply(temps, 2, function(x) na.omit(x))
# b.
sapply(temps.l, which.max)
```

```
##  City.A.July 9 City.B.July 29  City.C.July 6 City.D.July 14 City.E.July 20
##              8             29              6             13             15
##  City.F.July 4 City.G.July 16  City.H.July 8 City.I.July 21 City.J.July 18
##              3             15              6             18             18
```