Scientific Programming – Data processing in R

Exercises:

1) In the lecture we have seen what can happen with the row names of a `matrix` when using `rbind` to add a vector with or without a variable name …

- Try the same thing with a `data.frame`, i.e., use `rbind` to add one vector without an explicit variable name (e.g., `c(1, 2, 3, 4)`) and one vector with an explicit variable name (e.g., `newrow`). What do you observe? Is it like for matrices or do you make a different observation?
- Now try to add two columns to the `data.frame` using `cbind`: one without an explicit variable name and one with an explicit variable name … what do you observe now?
- [Note: always remember: `matrix` and `data.frame` are NOT the same thing, even if you fill them with exactly the same data!]

2) S3 class

- Create a list object `t1` with five elements named “patient” (numeric), “type” (string), “stage” (numeric), “age” (numeric), “metastases” (logical). If you want an example of some possible, tumor-related values, you can check the `data.frame` object we have seen in a previous lecture (but here we create a list object, not a data frame).
- Write out the content of the list object by simply typing: “`t1`”. Remember this output (e.g., copy and paste is somewhere for later comparison)
- Transform this list object to an object of S3 class “tumor”.
- Now write out the content of the object `t1` again. Do you observe any difference?
- Write a function `print.tumor` which takes an object of class “tumor” and prints out the clinical data in a human readable phrase. Make sure the function verifies the class before printing anything out. (Alternatively, remove the class check and try to call the function on some other object which is not of the correct type, e.g., some other list object from a previous lecture. You should get some error message because it doesn’t have the right data items/slots.) Hint: within this function, you can use for example the commands `print` and `cat` for printing strings, and `paste` and `paste0` for concatenating individual objects to strings. Check the online help for these functions.
- Test your function with `t1`.
- Now write a constructor for the class `tumor` and use it to generate a second object `t2`.
- Test `t2` with help of the function `print.tumor`.
- Now let’s experience some “magic”. What happens if instead calling
  ```
  print.tumor(t2)
  ```
  you call just
  ```
  print(t2)
  ```
  ?

  This happens because `print` is a so-called “generic” function. When called, R tries to look for a class-specific implementation named `print.<classname>` … if it cannot find such an implementation, it takes `print.default` instead. You can check which specific implementations of `print` are defined by the following command:
  ```
  methods(print)
  ```

Keep this in mind. Defining such class-specific implementations of generic functions may be useful for you own future projects.
Another important example of such a generic function is `plot`.

It is even possible to make your own generic function like `print` or `plot` but we will not cover this issue here. If you want to know more, see for example
https://www.datamentor.io/r-programming/S3-class/

3) S4 class
(For this exercise best start a new workspace so that the previous S3 class definition does not interfere with the following; alternatively, don't use the same name "tumor" for the new S4 class.)
Instead of using the S3 class system, define the `tumor` class using the S4 system.

- Create a tumor object `t3`. What does it look like when you print its content? Compare this to what you obtained above for an S3 class object.
- Although we don't need an explicit constructor for S4 classes – because we can use the `new` function to construct instances – it's a good idea to use the constructor which is automatically provided as return value from `setClass`. If you are interested in details, see for example: https://www.datamentor.io/r-programming/S4-class/
  See the same tutorial in case you want to know more about defining generic functions like `print` and `plot` for S4 classes.

4) Reference class: do the same as in task 3, but using a reference class instead of an S4 class.

5) Compute the standard deviation of a numeric vector using only the functions `sum`, `mean`, `length` and `sqrt`. Verify that your result is the same as obtained when directly using the function `sd`.

6) Define a numeric matrix `m1`

- Compute the z-scores of all elements
- Compute the natural logarithm and the logarithms for bases 2 and 10 of all elements
- Define a second numeric matrix `m2` of the same size and perform an element-wise multiplication of the two matrices
- Compute the true matrix multiplication
- Try what happens when you apply some comparison operation of the matrix, e.g.
  ```
  m1 > mean(m1)
  ```
  or
  ```
  m1 != 0
  ```
  (depending on what values you have in the matrix).
  Now try to use the outcome of the comparison operation for getting a subset of the matrix.

7) This exercise serves to to illustrate its superior efficiency of vectorized operations with respect to `for`-loops …

- Install (and then attach) the Bioconductor package "GEOquery"
- Have a look at its Bioconductor website to figure out what purpose the package serves
- Using the GEOquery function `getGEO`, obtain the data for the gene expression data set GSE102484 (see https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE102484)

```
# Download and load the breast cancer dataset GSE102484 from GEO
gse102484 <- getGEO('GSE102484',GSEMatrix=TRUE)
```

```
# convert expression data into a matrix
exprmat <- exprs(gse102484[[1]])
```

- How many rows and columns does the expression matrix have?
  (Note: this is a rather old gene expression data set obtained from Affymetrix microarrays; each matrix column is associated with a sample, each row with a microarray probeset; gene expression levels can be derived from probset measurements, but for this exercise probeset data is perfectly fine.)
- Create a second matrix `trmat` of the same size (initialized with NaNs) to contain transformed data, e.g., from a z-transformation (transformation to z-score) or log2-transformation.
- Use two nested `for`-loops to convert all individual expression scores to their corresponding log2-value, i.e., loop over all rows, and for each single row loop over all columns to convert the single elements of the matrix. Measure the time necessary to perform this operation on the thousands of rows and hundreds of columns. For measuring running time in R, the following tutorial may be of aid:
  https://www.r-bloggers.com/5-ways-to-measure-running-time-of-r-code/
  (especially points 1 and 3; an alternative to point 1 is the command `date`)
- Now do the same log2-transformation as a vectorized operation over the entire expression matrix: `log2(exprmat)`. Measure the time which is necessary for this operation.
- Is there any significant difference in the in the elapsed time? Which procedure is faster?

8) This exercise serves to learn the use of `apply` ...
- Use the same expression data matrix as above.
- Write a function which uses a for-loop for a column-wise z-transformation of the expression matrix (without using `apply`). Can you use this function also for computing row-wise z-scores?
- Now write a function for a vectorized z-transformation (i.e., z-scores for a vector) and use it with `apply` to perform a column-wise z-transformation of the expression matrix. Can you use the same function also for a row-wise z-transformation?

9) `...`
- Write a function which takes as arguments a vector `x`, a multiplication factor `a`, an offset `b`, and an exponent c and returns
  $$a * (\bar{x})^c + b$$
  (or write some other function taking additional arguments …)
- Apply this function both row- and column-wise to a numeric matrix and verify that the output has the shape you expected (dimensions!)

10) Using the following list
```
x <- list(a = 1:5, b = rnorm(10))
```
Use the function `exp` on all vector elements within this list. Which apply function will give you a simplified output, `lapply` or `sapply`? Test it. Can you explain the result?

11) Find other interesting examples for other `apply` functions (`lapply`, `sapply`, ...) online and try to repeat them.