Solutions - Class 2: Objects and classes

Koen Plevoets

```
library(methods)
```

1.

Create the following vectors with the function `rep()` (or its variants described on its help page):

- `3 3 3 3 3` using (only) the arguments `3` and `5`

```
rep(3, times = 5)
```

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

```
rep.int(3, times = 5)
```

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

- `3 3 1 1 2 2` using (only) the arguments `c(3, 1, 2)` and `2`

```
rep(c(3, 1, 2), each = 2)
```

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

- `3 1 2 3 1` using (only) the arguments `c(3, 1, 2)` and `5`

```
rep(c(3, 1, 2), length.out = 5)
```

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

```
rep_len(c(3, 1, 2), length.out = 5)
```

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

- `3 3 3 1 2 2` using (only) the arguments `c(3, 1, 2)` and `c(3, 1, 2)`

```
rep(c(3, 1, 2), times = c(3, 1, 2))
```

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

```
rep.int(c(3, 1, 2), times = c(3, 1, 2))
```

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

2.

Create the following vectors with the function `seq()` (or its variants described on its help page):

- `1 3 5 7 9`

```r
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```r
seq.int(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

- `1.0  5.5 10.0` using an argument `c(3, 2, 1)`

```r
seq(from = 1, to = 10, along.with = c(3, 1, 2))
```

```
## [1]  1.0  5.5 10.0
```

- `1.0  2.5  4.0  5.5  7.0  8.5 10.0` using an argument `7`

```r
seq(from = 1, to = 10, length.out = 7)
```

```
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

- `10  8  6  4  2`

```r
seq(from = 10, to = 1, by = -2)
```

```
## [1] 10  8  6  4  2
```

```r
seq.int(from = 10, to = 1, by = -2)
```

```
## [1] 10  8  6  4  2
```

3.

The function `replicate()` is sometimes wrongly used instead of `rep()` to repeat values. Use the function `system.time()` to measure the performance of both functions in repeating the value `3` for a total of `1e6` times.

```r
system.time(rep(3, 1e6))
```

```
##    user  system elapsed 
##       0       0       0
```

```r
system.time(replicate(1e6, 3))
```

```
##    user  system elapsed 
##    0.42    0.00    1.42
```

4.

The difference between `integer` and `double`(-precision floating point number) values is the byte size with which R stores them in memory. Byte size is also dependent on the attributes of the object. Use the function `object.size()` to show this for:

- the integer sequence `1:10`
- the numeric sequence `seq(1, 10, by = 1)`
- the 5x2 integer matrix `matrix(1:10, nrow = 5)`
- the 5x2 numeric matrix `matrix(seq(1, 10, by = 1), nrow = 5)`
- the integer vector `obj05 <- 1:10` of S3 class `humpty`
- the numeric vector `obj06 <- seq(1, 10, by = 1)` of S3 class `dumpty`

List these six objects in increasing order of byte size in your R code.

```r
# Creation of objects:
obj01 <- 1:10
obj02 <- seq(1, 10, by = 1)
obj03 <- matrix(1:10, nrow = 5)
obj04 <- matrix(seq(1, 10, by = 1), nrow = 5)
obj05 <- 1:10
class(obj05) <- "humpty"
obj06 <- seq(1, 10, by = 1)
class(obj06) <- "dumpty"

# Increasing order of byte sizes:
object.size(obj01)
```

```
## 96 bytes
```

```r
object.size(obj02)
```

```
## 176 bytes
```

```r
object.size(obj03)
```

```
## 264 bytes
```

```r
object.size(obj05)
```

```
## 320 bytes
```

```r
object.size(obj04)
```

```
## 344 bytes
```

```r
object.size(obj06)
```

```
## 400 bytes
```

5.

Rank the following objects/vectors according to the byte size. Which result surprises you?

- `rep(c(TRUE, FALSE), 5)`
- `seq(1, 10, by = 1)`
- `vector(mode = "logical", length = 10)`
- `vector(mode = "numeric", length = 10)`
- `vector(mode = "list", length = 10)`
- `vector(mode = "character", length = 10)`

```r
object.size(vector(mode = "logical", length = 10))
```

```
## 96 bytes
```

```r
# is as big as a "full" logical vector:
object.size(rep(c(TRUE, FALSE), 5))
```

```
## 96 bytes
```

```r
object.size(vector(mode = "numeric", length = 10))
```

```
## 176 bytes
```

```r
# is as big as a "full" numeric vector:
object.size(seq(1, 10, by = 1))
```

```
## 176 bytes
```

```r
# Both are as big as an empty list:
object.size(vector(mode = "list", length = 10))
```

```
## 176 bytes
```

```r
# Character vectors are biggest, however::
object.size(vector(mode = "character", length = 10))
```

```
## 232 bytes
```

```r
# Since empty vectors, which usually serve to allocate memory to "placeholders",
# take up as much memory as "full" vectors of the same length and type, your
# computer never has to re-allocate memory when "filling up" the empty vector.
# This is computationally efficient, provided you do not change type or length.
```

6.

Show the difference between `NULL` and `NA` by comparing their byte sizes. What does this say about a `NULL` object having attributes (Answer: read the help page of `NULL`)? Verify it by comparing the byte sizes of two names vectors `obj07 <- c(label = NULL)` and `obj08 <- c(label = NA)`.

```r
object.size(NULL)
```

```
## 0 bytes
```

```r
object.size(NA)
```

```
## 56 bytes
```

```r
obj07 <- c(label = NULL)
obj08 <- c(label = NA)

object.size(obj07)
```

```
## 0 bytes
```

```r
object.size(obj08)
```

```
## 280 bytes
```

7.

According to Euler's identity:

$$
\begin{aligned}
e^{i\pi} &= -1 \\
e^{i\pi} &= i^2 \\
ln(e^{i\pi}) &= ln(i^2) \\
i\pi\, ln(e) &= 2\, ln(i) \\
i\pi &= 2\, ln(i) \\
\pi &= \frac{2\, ln(i)}{i}
\end{aligned}
$$

Show that the right-hand side (of the last equation) indeed returns the left-hand side as the result in R.

```r
2 * log(0 + 1i) / (0 + 1i)
```

```
## [1] 3.141593+0i
```

8.

Create an S4 class `alphabetS4` and an RC class `alphabetRC`, both with slots/fields `symbols`, `size` and `type`. Instantiate both classes with the 26 letters of the alphabet (in lowercase) as values for `symbols`, the value `26` for `size` and the value `roman` for `type`.

```
setClass("alphabetS4",
         slots = c(symbols = "character", size = "numeric",
                   type = "character"))
setRefClass("alphabetRC",
            fields = c(symbols = "character", size = "numeric",
                       type = "character"))
obj09 <- new("alphabetS4", symbols = letters, size = length(letters),
            type = "roman")
obj10 <- new("alphabetRC", symbols = letters, size = length(letters),
            type = "roman")
```

9.

Create a list and and environment with the same components as the class instances of the previous exercise. Rank all four objects on the basis of their byte sizes.

```
obj11 <- list(symbols = letters, size = length(letters), type = "roman")
obj12 <- new.env()
obj12$symbols <- letters
obj12$size <- length(letters)
obj12$type <- "roman"

object.size(obj12)  # Environment
```

```
## 56 bytes
```

```
object.size(obj10)  # RC object
```

```
## 688 bytes
```

```
object.size(obj11)  # List
```

```
## 2320 bytes
```

```
object.size(obj09)  # S4 object
```

```
## 2736 bytes
```

```
# However, RC instances or environments can contain elements of a bigger size:
object.size(obj10$symbols)
```

```
## 1712 bytes
```

```
object.size(obj12$symbols)
```

```
## 1712 bytes
```

10.

Create:

- the vector `c(symbols = "a", size = "1", type = "roman")` called `vecX`
- the expression `vecY <- c(symbols = "a", size = "1", type = "roman")`

Evaluate the expression and verify that `vecX` is identical to the (newly created) object `vecY`. Determine the byte size of the expression.

```
vecX <- c(symbols = "a", size = "1", type = "roman")
expX <- expression(vecY <- c(symbols = "a", size = "1", type = "roman"))

eval(expX)

identical(vecX, vecY)
```

```
## [1] TRUE
```

```
object.size(expX)
```

```
## 1120 bytes
```

11.

Create a warning object `Watch out for this!` and print its attributes. What do you expect about the byte size of the warning object as compared to the byte size of the simple string `"Watch out for this!"`? Verify your expectation.

```
warX <- simpleWarning("Watch out for this!")
attributes(warX)
```

```
## $names
## [1] "message" "call"
##
## $class
## [1] "simpleWarning" "warning"       "condition"
```

```
object.size("Watch out for this!")
```

```
## 136 bytes
```

```
object.size(warX)
```

```
## 864 bytes
```

12.

Are condition objects such as errors or warnings recursive or atomic? You can use the warning object of the previous exercise to determine this.

```
is.atomic(warX)
```

```
## [1] FALSE
```

```
is.recursive(warX)
```

```
## [1] TRUE
```