# Lecture 1: solutions

Statistical Computing with R

## Exercise 1

### 1.

We can use the `%%` operator to compute the remainder. The `%/%` can be used to compute the integer part of a division.

```
# Remainder
12 %% 5
```

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

```
# Integer part
12 %/% 5
```

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

### 2.

```
# Remainder
823 %% 17
```

```
## [1] 7
```

```
# Integer part
823 %/% 17
```

```
## [1] 48
```

### 3.

```
# Remainder
475832 %% 1342
```

```
## [1] 764
```

```r
# Integer part
475832 %/% 1342
```

```
## [1] 354
```

## Exercise 2

### 1.

```
37 * 42 + 17 / 3
```

```
## [1] 1559.667
```

### 2.

```
17 / 3 - 2 * (15 / 6 + 1)
```

```
## [1] -1.333333
```

### 3.

You can use both ^ and ** to indicate powers. Both solutions are thus identical.

```
(3 * (3 / 4)**2 - 5) / 2 + 7
```

```
## [1] 5.34375
```

```
# alternatively
(3 * (3 / 4)^2 - 5) / 2 + 7
```

```
## [1] 5.34375
```

## Exercise 3

The composite interest rate over a certain number of years $n$, can be computed as

$$(\prod_{i=1}^{n}(1+r_i))-1$$

where $r_i,\ i=1,...,n$ denotes the yearly interest rates.

```
# Interbank
ib <- 1.015^5 - 1

# PiggyBank
pb <- 1.013**3 * 1.02**2 - 1

# compare
c(ib, pb)
```

```
## [1] 0.07728400 0.08150537
```

In the first two lines we calculated how much interest we would get at Interbank and PiggyBank. To calculate powers you can either use `^` or `**`. We save both answers in the variables ib and pb respectively. The third line prints the outcomes of both calculations. Since $pb > ib$, Fernanda should choose Piggybank for her deposits.

## Exercise 4

The `seq` command is a function that takes three arguments. First a starting position `from`, then an end position `to`. Lastly, the step increment can be specified with the argument `by`. By adjusting these arguments you can create any sequence between two numbers by a specific increment.

### 1.

The sequence we want is from 1 to 8 in increments of 1. So we first specify the starting position 1, then the end position 8, and lastly we specify the step length, in this case 1. You do not have to specify which variable is which specific value. For a lot of functions R knows the order of the variables. However, sometimes it might be more clear to write the variables out, especially on projects with other people or in selfmade functions.

```
x <- seq(from = 1, to = 8, by = 1)
x
```

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

```
# alternatively
x2 <- seq(1, 8, 1)
x2
```

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

```
# for these simple cases with increments of 1 you can also use
x3 <- 1:8
x3
```

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

### 2.

Similarly to the last exercise we specify the start, end, and increment. However, this time we start at 5, end at 17, and have an increment of 3.

```
y <- seq(from = 5, to = 17, by = 3)
y
```

```
## [1]  5  8 11 14 17
```

### 3.

The answer is similar to the previous ones. Remember, `R` uses a dot to specify decimals, not a comma.

```
z <- seq(from = 4.5, to = 10.5, by = 2)
z
```

```
## [1]  4.5  6.5  8.5 10.5
```

# Exercise 5

## 1.

The function `rep` (from replicate) replicates some input (could be a single number, a string, a vector, etc) and replicates this a set amount of `times`. In this case we want to replicate the vector c(14, 15) three times to get the result (14, 15, 14, 15, 14, 15).

```
x <- rep(x = c(14, 15), times = 3)
x
```

```
## [1] 14 15 14 15 14 15
```

## 2.

This solution is similar to the previous one, however, notice that we have a vector instead of a single value for the times variable. The rep function produces the result by first taking the number at the first index of the x vector (2) and then replicates this by the number on the first index of the times vector (3). This results in three twos. Then it moves to the second index of the x vector (3), and the second index of the times vector (2) and we get two threes. This goes on until the end of the x vector.

```
y <- rep(x = c(2, 3, 1, 4), times = c(3, 2, 5, 2))
y
```

```
##  [1] 2 2 2 3 3 1 1 1 1 1 4 4
```

## 3.

As is usually the case when programming, problems often have multiple solutions. In this case we could code it just like we did in the previous exercise. However, you could also choose to create parts of the vector separately and combine them later in one vector, as is the case in alternative 1. Maybe you also noticed that male and female groups alternate. In alternative 2 we first create a vector of three times male and female repeated, and then use this as an input to specify how many times we want each value in the vector.

```
z <- rep(c('male', 'female', 'male', 'female', 'male', 'female'), c(3, 1, 2, 5, 1, 2))
z
```

```
##  [1] "male"   "male"   "male"   "female" "male"   "male"   "female" "female"
##  [9] "female" "female" "female" "male"   "female" "female"
```

```
# alternative 1
z1 <- rep(c('male', 'female'), c(3, 1))
z2 <- rep(c('male', 'female'), c(2, 5))
z3 <- rep(c('male', 'female'), c(1, 2))
z <- c(z1, z2, z3)
z
```

```
##  [1] "male"   "male"   "male"   "female" "male"   "male"   "female" "female"
##  [9] "female" "female" "female" "male"   "female" "female"
```

```r
# alternative 2
sex <- rep(c('male', 'female'), 3)
z <- rep(sex, c(3, 1, 2, 5, 1, 2))
z
```

```
##  [1] "male"   "male"   "male"   "female" "male"   "male"   "female" "female"
##  [9] "female" "female" "female" "male"   "female" "female"
```

## Exercise 6

A way to check which data type your variable is, is to manually go by all data types and check if it returns `TRUE` or `FALSE`. `v1` returns `TRUE` for `is.numeric(v1)`, thus `V1` is numeric. Instead of guessing you could also use the function `class()`, which immediately returns the data type.

```
# specifying the variables
v1 = c(exp(3), 2*pi)
v2 = c('Antonio', v1)
v3 = factor(c('green', 'green', 'red'))
v4 = c('green', 'green', 'red')
```

```
# Checking data type
is.numeric(v1)
```

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

```
is.character(v1)
```

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

```
is.factor(v1)
```

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

```
# Alternatively
class(v1)
```

```
## [1] "numeric"
```

```
is.character(v2)
```

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

```
is.factor(v3)
```

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

```
is.character(v4)
```

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

# Exercise 7

## 1.

You can either immediately input a vector into the `sum()` function, or first create the variable, and then input it.

```
x <- 1:100 # you could also use x <- seq(1, 100, 1)
# but 1:100 is default for R for creating a sequence between
# the left and right side with an increment of 1
sum(x)
```

```
## [1] 5050
```

```
# Alternatively
sum(1:100)
```

```
## [1] 5050
```

## 2.

```
sum(3:98)
```

```
## [1] 4848
```

## 3.

```
sum((5:10)^2)
```

```
## [1] 355
```

```
# Alternatively
x <- 5:10
x2 <- x^2
sum(x2)
```

```
## [1] 355
```

## 4.

`R` is smart with vector operations (one of `R`s strengths), you can do direct arithmetic with vectors. Just like `7:14` would create a vector containing the numbers $7, 8, ..., 13, 14$, you can create a vector containing $\frac{1}{7}, \frac{1}{8}, ..., \frac{1}{13}, \frac{1}{14}$ using `1 / 7:14`. This can be used in the `prod()` (product) function.

```r
x <- 7:14
x_inv <- 1 / x
prod(x_inv)
```

```
## [1] 8.258937e-09
```

```r
# Alternatively
prod(1 / 7:14)
```

```
## [1] 8.258937e-09
```