Beginner R exercises

B 1) The code created two variables (**a** and **b**) in your environment. The value of variable **a** is 2 (although at first it is specified as 1) and the value of variable **b** is 1. It does not matter whether variables are assigned with the **=** or the **<-** operator.

B 2)
```
rm(a); rm(b)
```

B 3) Adding up **apples** and **pears** equals 11.
The variable **Pears** does not exist. Instead, you should add **apples** and **pears** together (notice the lack of a capital P).

B 4)
```
t1 <- sqrt(81)
```

B 5)
```
t2 <- 81^0.5
```

B 6)
```
t1 == t2
```

B 7) The working directory is a file path on your computer that sets the default location of any files you read into R, or save out of R. In other words, a working directory is like a little flag somewhere on your computer which is tied to a specific analysis project. If you ask R to import a dataset from a text file, or save a data frame as a text file, it will assume that the file is inside of your working directory.

B 8) You can change the working directory using the **setwd()** function. You can check whether your change worked by calling the **getwd()** function again and checking whether it is now at the folder you specified in **setwd()** . Note that RStudio also has the option to change the working directory from the "Session" dropdown menu, then select "Set Working Directory".

B 9) The code shows all the available demos that are built into R. The **persp** demo can be viewed with:

```
demo(persp)
```

B 10) It gives `NA` because the `mean()` function does not handle `NA`'s by default. You can compute the mean without the missing value by setting `na.rm = TRUE`.

B 11) Just typing `mvrnorm()` gives an error because the package `MASS` (from which the function comes) is not loaded yet. You can load the `MASS` package by typing:

```
library(MASS)
```

B 12) The `%%` operator gives the modulo (the remaining number after division) of the first number divided by the last number. You cannot find help by typing `?%%`, but you can search Google for help on how to use this operator. Note that in RStudio you can also type `%%` into the search bar in the Help section (in the Files and Plots part of RStudio, usually displayed at the bottom right of your screen).

B 13) You can test the correlation and find the confidence interval for the correlation by using the `cor.test()` function. You can find all function that have 'cor' in their name by typing `apropos('cor')`.

```
cor.test(c(1, 2, 3, 4), c(1, 4, 7, 15))
```

B 14) The modes of `a1`, `a2`, and `a3` are character, numeric, and logical respectively. The modes of `b1`, `b2`, `b3`, and `b4` are character, character, numeric, and character respectively. When vectors of different modes are combined R converts the vectors to one and the same mode, because elements in a vector can only be of one mode.

B 15) As a standard, R converts `TRUE` to a 1 and `FALSE` to a 0.
$1/1 = 1$ $0/1 = 1$ $0/0 =$ undefined $1/0 = \infty$

B 16) In the first case the numeric `1` is converted to character mode, while in the last case the logical `TRUE` is converted to character mode.

```
as.numeric(TRUE)
as.character(TRUE)
```

B 17) You can check whether a vector is numeric by using the `is.numeric()` function. `c(1, 0)` is numeric; `c(TRUE, FALSE)` is not; `c(TRUE, FALSE, 1, 0)` is numeric because R automatically converts `TRUE` and `FALSE` to a `1` and a `0` because vectors have to be of one mode.

B 18) The length of this vector is 11.

```
v1 <- c(-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8)
# or
v1 <- -2:8
length(v1)
```

B 19)

```
v2 <- seq(from = -5, to = 5, by = 0.5)
v2 <- (-10:10)/2
```

B 20)

```
v3 <- seq(from = 13, to = 33, by = 2)
```

B 21)

```
c(a, b)
```

B 22) The result of `rep(2, 10)` is a repeated vector containing 10 times the number 2. The function argument `times = 10` is implicitly set.

B 23)

```
rep(3:7, each = 3)
```

B 24) You have to explicitly specify that the 2 refers to the `each` argument. By default, the second argument of the `rep()` function is times, see also `?rep` .

B 25) The sum of these numbers is 100.

```
sum((1:10)*2-1)
```

B 26)

```
v3 <- (1:10) * 2
v4 <- v3 / 5
v5 <- seq(5, 32, 3)
```

B 27)
```
vector('logical', 5)
```

B 28)
```
paste(rep(c('x', 'y'), each = 4), rep(1:2, each = 2), c('m', 'f'), sep = '')
```

B 29) The options `decreasing = TRUE` can be set to sort the numbers from highest to lowest. The default is `FALSE`, so sorting from lowest to highest does not require specification of this option.

```
sort(s, decreasing = TRUE) # Highest to lowest
sort(s) # Lowest to highest
```

B 30) R is actually pretty smart and will recycle values for you. 10 is not divisible by 3, hence full recycling of all numbers is not possible.

B 31)
```
matrix(25:1, nrow = 5, byrow = TRUE)
```

B 32)
```
matrix(rep(0:1, 8), nrow = 4)
```

B 33) You do not have to specify the number of rows (only the number of columns) because, given the number of values and the number of columns, the number of rows for the matrix is known.

B 34) You can transpose a matrix using the `t()` function.

B 35)
```
colMeans(m1)
```

B 36) You can select the diagonal of a matrix using the `diag()` function.

```
m2 <- matrix(rep(0, 16), nrow = 4)
diag(m2) <- diag(m1)

# or:
m2 <- diag(diag(m1))
```

B 37)
```
m1 <- rbind(m1, rowSums(m1))
```

B 38) The means of the columns are all zero (calculate with rowSums(m1)), and the standard deviations are all one (calculate with apply(m2,2,sd). The **scale()** function therefore transforms the values in the columns so that their mean is zero and their standard deviation is one.

B 39)
```
data.frame(subject = rep(1:5, each = 2),
           time = c('t1', 't2'),
           score = c(7, 8, 8, 8, 9, 8, 9, 7, 7, 6))
```

B 40)
```
as.data.frame(a)
```

B 41)
```
list(subject = 1:5,
       time = c('t1', 't2'),
     score = matrix(1:25,5,5))
```

B 42) No, **g** is not a true vector but a factor. That is because **g** consist of factor levels. You can check this using **is.vector(g)** or **is.factor(g)** . Note that you can also find this out by using the 'str' function: str(g).

B 43)
```
v <- as.factor(v)
```

B 44) The factor **x** has three levels, which you can find out using the **levels()** function.

```
x <- as.factor(x)
levels(x)
```

B 45) The value of `x[2]` is `NA`.

B 46)
```
a[1] <- 8
a[a == 2] <- 0
```

B 47)
```
l[2, 3]
```

B 48)
```
l[c(3, 5), ]
```

B 49)
```
m[m[, 1] < 5, ]
```

B 50)
```
m[ , -4]
```

B 51)
```
m <- as.data.frame(m)
colnames(m) <- paste('trial', 1:10, sep = '.')
```

B 52)
```
m$'trial.1'[2:5]
m$'trial.4'[1:2]
```

B 53)
```
b <- ifelse(b == 1, yes = 2, no = 1)
```

B 54)
```
gsub('a', '.', n)
```

B 55)
```
passed <- grades[rowMeans(grades[, c('exer','exam')]) > 5.5 &
                 grades[, 'exer'] >= 5 &
                 grades[, 'exam'] >= 5 , 1]
failed <- grades$student[-passed]
```

B 56) The condition `(a < 0 | b < 0) (a * b < 0)` means that `a` is smaller than `0` or `b` is smaller than `0`, and `a * b` is smaller than `0`. If this condition is `TRUE` and `a` is negative, then `b` must be positive.

B 57) The number of `NA`'s in the vector is 1, hence it is smaller than or equal to 2. So, `sum(is.na(c(1, 2, 3, 4, NA, 7))) <= 2` returns `TRUE`, and its logical negation (`!`) is therefore `FALSE`.

B 58) These are the numbers by which the number 24 can be divided (24 is divisible by 1, 2, 3, 4, 6, 8, 12, and 24).

B 59) This code tests whether two vectors are identical. The `identical()` function does this automatically.

```
min(x==y) == 1
mean(x==y) == 1
prod(x==y) == 1
identical(x, y)
```

B 60) You can use the `table()` function to create a frequency table of the throws.

B 61)
```
sample(c('jack', 'queen', 'king', 'ace'), replace = TRUE)
```

B 62) You can use the `runif()` function to sample uniform random numbers. `sample(1:100, 20)` is not correct in this case because the sampling vector starts at 1.

```
runif(n = 20, min = 0, max = 100)
```

B 63)
```
r1 <- runif(n = 21, min = 0, max = 100)
median(r1)
r2 <- sort(r1, decreasing = TRUE)
r2[11]
```

B 64) Simulating data using the same code twice does not result in the same samples because of the inherent randomness of a simulation. You can use the **set.seed()** function to make your code reproducible.

```
set.seed(123)
r3 <- rnorm(n = 100, mean = 100, sd = 15)
var(v3)
```

B 65)

```
cov(x)
cor(x)
```

B 66) The **read.csv()** function reads in **.csv** files. You can read in Excel files (**.xlsx**) using the **read.xslx()** function (from the **xlsx** package). The function **read.spss()** is featured in the package **foreign** (see **?read.spss**) and reads **.spss** files.

B 67)

```
# Be sure to set your working directory when providing a relative path
dataset <- read.csv('example.csv')
```

B 68)

```
# Be sure to set your working directory when providing a relative path
write.csv(d, file = 'example.csv')
```