SCR week 1: exercises

Exercises part 1: Objects and types, using functions

1.1 Working with the work space

`R`'s workspace is a nonphysical 'environment' that contains (remembers) the variables that we construct in our `R` commands. This workspace seems empty as we start `R` up. In this exercise we will look at some aspects of `R`'s workspace and scripts.

a.

Create two variables in `R`'s console, called `T` and `Y`. Assign the values `5` and `20` respectively.

Answer:

```
T <- 5
Y <- 20
```

b.

Write a script that multiplies the values of `T` and `Y`.

Answer:

```
T * Y
```

```
## [1] 100
```

c.

Save the script file, with the name `Separated.R`.

Answer:

Go to File -> Save As.

d.

Close Rstudio, and start it up again. Make sure the scriptfile `Separated.R` is loaded into Rstudio (which it will be by default if you did not explicitly close the script before you exited Rstudio, and don't save the workspace).

e.

Use the script to run the multiplication between `T` and `Y` again. Does it work? Why not?

If done correctly, you will notice that `Y` not longer exists, and that `T`, after you restart `R`, will be synonymous again with `TRUE`. Like a few other objects, `T` is put in the workspace of `R`, automatically, but can as we've witnessed be overwritten. In general it is not a good idea to overwrite existing object names, such as `T` or `F`: if you combine your code with somebody else's they might have use `T` instead of `TRUE` in their code to check if something is true or not. If you overwrite `T`, their code will not work anymore. Similarly: it is better to use `TRUE` than `T`.

1.2 Coercion

We've seen three modes (or data types): numeric, character and logical. We've also seen that `R` will sometimes automatically convert one type, into another, if it thinks that's what you want it to do. For example, if we multiply `TRUE` by 10, the answer is 10. This is called *implicit coercion*: `TRUE` is coerced, or forced, to be interpreted as a `1`.

a.

Try multiplying some numbers with `TRUE` and `FALSE` yourself.

Answer:

```
TRUE * 1
```

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

```
TRUE * 10
```

```
## [1] 10
```

```
FALSE * 1
```

```
## [1] 0
```

```
FALSE * 10
```

```
## [1] 0
```

b.

Take the following character values and assign them to some objects (give some sensible names yourself): "The number two", "2" and "two". Use the function `mode` to check if the data type of entries is `character`, `logical` or `numeric`.

Answer:

```
a <- "The number two"
b <- "2"
c <- "two"
mode(a); mode(b); mode(c)
```

```
## [1] "character"

## [1] "character"

## [1] "character"
```

c.

Try multiplying the objects you've created by 10. Do you get a warning, or worse, an error?

Answer:

```
a * 2
```

```
## Error in a * 2: non-numeric argument to binary operator
```

d.

Unfortunately, this does not work (`R` gives an error). In this case `R` does not automatically coerce the 3 objects to numbers. We can force `R` to try to coerce the objects to a numeric one using the function `as.numeric`. Apply `as.numeric` to the objects you've created. Do you get an error, or a warning?

Answer:

```
as.numeric(a)
```

```
## Warning: NAs introduced by coercion

## [1] NA
```

```
as.numeric(b)
```

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

```
as.numeric(c)
```

```
## Warning: NAs introduced by coercion

## [1] NA
```

e.

We got a warning because `R` does not know how to convert `"The number two"` or `"two"` to a number: instead it turned those objects to `NA` which stands for `Not Available` and can be considered a 'missing' value. Amazingly however, `R` does know how to convert `"2"`, to 2! Try it the other way around by converting a few numbers to characters, by using the `as.character` function. Does this produce any `NA`'s?

Answer:

```
as.character(1:10)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

f.

Take the values `-3`, `-1`, `0`, `1` and `1000` and coerce each to the logical type, by using the function `as.logical`. What is the result?

Answer:

```r
as.logical(c(-3, -1, 0, 1, 1000))
```

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

It seems that 0 is converted to `FALSE` and all other values to `TRUE`!

g.

What do you think will happen in the following call: `as.character(TRUE)` And in: `as.logical("TRUE")`, or `as.logical("completely false")`?

Answer: Just run the code, you'll see.

Exercises part 2: vectors and functions

2.1 Operations on vectors

a.

Create a vector containing the values 0.2, 0.4, 0.6, . . . 1.8, 2.0.

Answer:

```
my_vec <- (1:10)/5
```

b.

Go to the vocabulary that's put online by Hadley Wickham: http://adv-r.had.co.nz/Vocabulary.html. Look at the operators under the **basic math** header. Try a few of the following operators:

*, +, -, /, ^, %%, %/% abs, sign acos, asin, atan, atan2 sin, cos, tan ceiling, floor, round, trunc, signif exp, log, log10, log2, sqrt

On which element(s) of the vector `my_vec` does the function operates?

Answer:

```
my_vec * 2
```

```
##  [1] 0.4 0.8 1.2 1.6 2.0 2.4 2.8 3.2 3.6 4.0
```

```
sign(my_vec)
```

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

```
acos(my_vec)
```

```
## Warning in acos(my_vec): NaNs produced
```

```
##  [1] 1.3694384 1.1592795 0.9272952 0.6435011 0.0000000       NaN       NaN
##  [8]       NaN       NaN       NaN
```

```
signif(my_vec)
```

```
##  [1] 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

These functions all work on the individual elements of the vector. These functions are *vectorized*.

c.

The vocabulary also lists the following operators:

max, min, prod, sum cummax, cummin, cumprod, cumsum, diff pmax, pmin range mean, median, cor, sd, var rle

Try a few of these. Look at the helpfiles of a function (using e.g. `?max`) if you don't know what the function does. What is the big difference between these operators and the ones you tried in **b.**?

Answer:

```
max(my_vec)
```

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

```
range(my_vec)
```

```
## [1] 0.2 2.0
```

```
cumsum(my_vec)
```

```
##  [1]  0.2  0.6  1.2  2.0  3.0  4.2  5.6  7.2  9.0 11.0
```

These all perform some aggregate function: they take all of the elements in the vector to produce a single return value, this is very different from vectorization!

2.2 Creating a typical function

You've seen how to create a function:

```
FunctionName <- function(argument){

  # do stuff

  return(return_value)
}
```

In this exercise we will walk through the typical process one might go through in create a function and look a the concept of functions and *scoping*.

a.

Create a vector using the following code: `my_vector <- c(4, 70, 19, 21, 77, 82, 75, 33, 90, 34, 6, 27, 63, 25, 39, 83, 42, 60, 17, 10)`.

Answer:

```
my_vector <- c(4, 70, 19, 21, 77, 82, 75, 33, 90, 34, 6, 27, 63, 25, 39, 83, 42, 60, 17, 10)
```

b.

Find the minimal value of the vector using the function `min`.

Answer:

```
min(my_vector)
```

```
## [1] 4
```

c.

Find the maximal value of the vector using the function `max`. **Answer:**

```
max(my_vector)
```

```
## [1] 90
```

d.

Add the maximum and minimum value together, divide by 2 and substract that value from the `mean` of the vector. Which is bigger?

Answer:

```
mean(my_vector) - (max(my_vector) + min(my_vector))/2
```

```
## [1] -3.15
```

e.

Write code to have `R` tell you whether it is `TRUE` or `FALSE` that the mean is bigger than the 'halfway' value of the range of our vector.

Answer:

```
my_difference <- mean(my_vector) - (max(my_vector) + min(my_vector))/2
my_difference > 0
```

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

f.

Write a function called `IsMeanBiggerThanHalfway` that takes as argument a vector and returns `TRUE` or `FALSE` depending on whether the mean is bigger that the halfway value of the range of the vector that is entered as argument.

Answer: We simply put the lines of code we wrote earlier, and wrap a function around it!

```
IsMeanBiggerThanHalfway <- function(a_vector_argument){

  my_difference <- mean(my_vector) - (max(my_vector) + min(my_vector))/2

  return(my_difference > 0)
}

IsMeanBiggerThanHalfway(my_vector)
```

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

Suppose we would have asked you to write a function that checks whether the mean of a given vector was larger than the halfway value of the range of that same vector. That would have involved doing all of the above steps. Hopefully, you've noticed that the steps we take are as follows: we write each step we have to take in sequence, so that we can check whether each step is written correctly or not. We also started out with a vector as an example to check if our code works. Only at the *very end* of the process, we turned the code into a function. This is a very important lesson in coding.

2.3 Creating a function with multiple arguments

You'll often see functions in `R` that can take multiple arguments. The result of the function (usually the return value) will depend on *both* arguments. `sd` is a function that takes two arguments. Without looking exactly how `sd` works, look at the helpfile of the `sd` en try to figure out from the syntax presented in the helpfile (under **Usage**) how to create a function that takes 3 arguments.

Write a function (choose an active name yourself) that takes three arguments. Let the function return the product of the three values.

Answer:

```
MultiplyThreeNumbers <- function(a, b, c){
  product <- a * b * c
  return(product)
}
```

2.4. Some Vector Exercises Again (slightly more difficult, for now)

While using `rep()`, `seq()` and/or arithemtic thinking, generate the following sequences:

(a) $10, 8, 6, 4, 6, 8, 10$

(b) $60, 56, 52, \ldots, 12, 8.$

(c) $1, 2, 4, 8, \ldots, 512$

(d) $0, 1, 2, 0, \ldots, 2, 0, 1, 2$ (with each entry appearing six times)

(e) $1, 2, 2, 3, 3, 3, 4, 4, 4.$

(f) $1, 2, 5, 10, 20, 50, 100, \ldots, 5x10^3$ (use vector recycling!)

Answer:

```r
#a:
abs(seq(-6, 6, by = 2)) + 4
```

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

```r
#b:
seq(60, 8, by = -4)
```

```
##  [1] 60 56 52 48 44 40 36 32 28 24 20 16 12  8
```

```r
#c:
2^(0:9)
```

```
##  [1]   1   2   4   8  16  32  64 128 256 512
```

```r
#d:
rep(0:2, 6)
```

```
##  [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

```r
#e:
rep(1:4, times = 1:4)
```

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

```r
#f:
c(1, 2, 5) * 10^(rep(0:3, each = 3)) # uses vector recycling!!!
```

```
##  [1]    1    2    5   10   20   50  100  200  500 1000 2000 5000
```

2.5. Don't Stop 'til You Get Enough (more difficult, for now)

While using `cos()`, exp(), `%%`, `:`, and/or arithemtic thinking, generate the following sequences:

(a) $cos\left(\frac{\pi n}{3}\right),$ for $n = \{0, \ldots, 10\}$.

(b) $1, 9, 98, 997, \ldots, 999994$.

(c) $e^n - 3n,$ for $n = \{0, \ldots, 10\}$.

(d) $3n \bmod 7,$ for $n = \{0, \ldots, 10\}$.

(e) Let

$$\tilde{\pi}_n = 4\sum_{i=1}^{n} \frac{(-1)^{i+1}}{2i-1} = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots + 4\frac{(-1)^{n+1}}{2n-1}.$$

Create a function 'ApproxPi' that outputs π_n when you give it n. You may want to use a 'sum()' function on a vector, and you could use vector recycling. Evaluate $\tilde{\pi}_{100}$, $\tilde{\pi}_{1000}$, and $\tilde{\pi}_{10000}$, do you notice anything?

Answer:

```r
# a:
cos(pi * (0:10)/3)
```

```
##  [1]  1.0  0.5 -0.5 -1.0 -0.5  0.5  1.0  0.5 -0.5 -1.0 -0.5
```

```r
# b:
10^(0:6) - 0:6
```

```
## [1]      1      9     98    997   9996  99995 999994
```

```r
# c:
exp(0:10) - 3 * (0:10)
```

```
##  [1]     1.0000000    -0.2817182     1.3890561    11.0855369    42.5981500
##  [6]   133.4131591   385.4287935  1075.6331584  2956.9579870  8076.0839276
## [11] 21996.4657948
```

```r
# d:
(3 * (0:10)) %% 7 # be careful not to forget the brackets!
```

```
##  [1] 0 3 6 2 5 1 4 0 3 6 2
```

```r
# e:
ApproxPi <- function(n) {
  # num <- (-1)^(1:n + 1)
  num <- c(1,-1) # when using vector recycling
  denom <- seq(1, 2*n - 1, by = 2)
  Sn <- sum(num/denom)
  return(Sn)
}
4 * c(ApproxPi(n = 100), ApproxPi(n = 1e3), ApproxPi(n = 1e4)) # it converges to pi for higher n..
```

```
## [1] 3.131593 3.140593 3.141493
```

Exercises part 3: Conditions and if, indexing and filtering

3.1. An absolute function

In this exercise we will create a function that returns the absolute difference between two values. Say we wish to find the absolute difference of the expression $4 - 10$ is equal to 6. We could use the following statement to find the absolute difference between `x` and `y` regardless of which is bigger:

```
abs(x-y)
```

Regardless of whether the result is positive or negative, that `abs` function will make it a positive number. We'll implement a function ourselves using conditions and an `if` statement.

a.

Write a line of code that substracts `y` from `x` and saves it as a new value `z`. To test your code you will need to assign some values to `x` and `y`.

Answer:

```
x <- 10
y <- 4
z <- x-y
```

b.

Write a condition (or logical expression) that checks whether `z` is negative.

Answer:

```
z < 0
```

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

c.

Write an `if` statement that uses the condition you've written above to check whether `z` is negative, and if so, multiplies `z` by -1 to make it positive.

Answer:

```
if (z < 0){
  z <- z * -1
}
```

d.

Put the above code into a function called `AbsoluteDifference` and test the code with the following value pairs:

- $x = 10$, $y = 4$

- $x = 4$, $y = 10$
- $x = 4$, $y = -10$

Did your function test correctly?

Answer:

```
AbsoluteDifference <- function(x, y){
  z <- x-y

  if (z < 0){
    z <- z*-1
  }

  return(z)
}

AbsoluteDifference(10, 4)
```

```
## [1] 6
```

```
AbsoluteDifference(4, 10)
```

```
## [1] 6
```

```
AbsoluteDifference(-10, 4)
```

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

3.2. Our first filter

We've seen during class that we can index in a variety of ways: with positions, with negative positions, with names and with logicals. In this exercise we'll be using logicals to create a filter.

a.

Create a vector, called short_alphabet, containing the first 10 (no capital) letters of the alphabet.

Answer:

```
short_alphabet <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
```

b.

Use brackets (`[]`) to select the 7th letter.

Answer:

```
short_alphabet[7]
```

```
## [1] "g"
```

c.

Besides giving the position (or positions!) of the elements you want to access you can also tell `R` which elements you **do** and which elements you **don't** want to select, by telling `R` for each position whether it is `TRUE` or `FALSE` that you want to select each element.

For example, we can select the second and fourth element of the vector `c(1, 2, 3, 4, 5)` in the following way:

```
a <- c(1, 2, 3, 4, 5)
a[c(FALSE, TRUE, FALSE, TRUE, FALSE)]
```

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

Try this yourself by creating a vector, containing only `TRUE` and `FALSE`, that you can use to select the 7th letter from the `short_alphabet` object.

Answer:

```
short_alphabet[c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE)]
```

```
## [1] "g"
```

d.

Instead of manually typing a vector of `TRUE` and `FALSE` we can use `R` and its vectorized functions to create one for us. In the previous exercise we've learned that `R` can evaluate a logical expression to `TRUE` or `FALSE`. It can do this in a vectorized manner. An example of a vectorized function is multiplication: if you have a vector of numbers, you can multiply the vector object with a constant, to multiply *all elements* in the vector with that number. For example:

```
a <- c(1, 2, 3, 4, 5)
a * 2
```

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

We can do a similar thing with a condition that checks for equality:

```
short_alphabet == "a"
```

Use this to create a vector that has a `TRUE` only in the 7th position. Save this to an object called `seventh_letter`.

Answer:

```
seventh_letter <- short_alphabet=="g"
```

e.

Create another vector with only `TRUE` and `FALSE`, but one with just a `TRUE` in the position of the letter 'c'. Save it to an object called `third_letter`.

Answer:

```
third_letter <- short_alphabet=="c"
```

f.

Our first 'filter' will be created by combining the two vectors containing only `TRUE` and `FALSE`. We won't skip ahead just yet and talk about more advanced parts of *control* statements. Instead, we will use a trick. We've already seen that we can use `TRUE` and `FALSE` for calculation: if forced to be read as a number,`TRUE` is equal to 1, and `FALSE` is equal to 0. Thus, if we add, pairwise, the elements of both vectors everything that was `FALSE` in both cases will be `0`, and everything that was `TRUE` in either or both will be `1` or `2`. If we coerce anything but `0` to a logical, `R` will make it `TRUE`. Thus we can add one vector to the other, coerce it to a logical vector and use it to subset (or index) our vector of numbers. Do this now for the letters `c` and `g`.

Answer:

```
short_alphabet[as.logical((short_alphabet=="g") + (short_alphabet=="c"))]
```

```
## [1] "c" "g"
```

3.3 Subsetting

Create a vector `x` of normal random variables as follows:

```
set.seed(123)
x <- rnorm(1000)
```

The `set.seed()` fixes the random number generator so that we all obtain the same `x`; changing the argument `123` to something else will give different results. This is useful for replication.

(a) show the first 10 elements from the vector 'x'

(b) show each 100th element of 'x'

(c) show how many of the elements are > 1 or < -1

(d) show the proportion of elements higher than 1.645

Answer:

```
set.seed(123)
x <- rnorm(1000)
# a:
x[1:10]; head(x, 10)
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##  [6]  1.71506499  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##  [6]  1.71506499  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

```r
# b:
x[(1:10)*100]
```

```
##  [1] -1.0264209 -1.1854801  1.2499146 -0.3545424  0.5521577  1.8599109
##  [7] -1.6140395  0.6475134  0.2435327 -0.2491907
```

```r
# c:
sum(x > 1 | x < -1)
```

```
## [1] 322
```

```r
# d:
mean(x > 1.645)
```

```
## [1] 0.056
```