STENO Introductory R-workshop: Variables, data types and containers

Leon Eyrich Jessen, leyj@steno.dk

June 9, 2015

Contents

1 Introduction

This workshop document is aimed at people with very little or no knowledge of R. We will start from absolute scratch focusing on programming concepts, such as variables, data types and containers.
Before we begin the actual learning part of this vignette, I want to stress one thing, that I find extremely important for you to know: R is difficult to learn! Especially if you are not familiar with programming. HOWEVER if you can mobilise a bit of **stubbornness** and **I-will-hang-in-there attitude**, your effort will be generously rewarded by R!

1.1 What is R?

"R is a free software environment for statistical computing and graphics" [http://www.r-project.org/]. So basically R is an advanced calculator, allowing you to conduct various calculations, analyses and visualisations.
In the following, there will be a mix of text like the one you are reading right now and every now and then, you will see actual R code in a grey box, like this:

```
# Calculate the sum of 2 and 3
2 + 3

## [1] 5
```

Where the first line beginning with a single '#' is the comment line. The second line is the actual R-code itself and the last line beginning with '##' is the result of running the R-code.
In this case we want to calculate the sum of $2 + 3$, so we state that in the comment, then follows the R-code, which simply is `2+3` and then the result 5 (We skip the [1] for now).

Note! Comments are important! They are used for explaining what you are doing, so that others may read your code or more importantly, you will know what is going on, when you have to take a look at the R-script you wrote 2 years ago.

1.2 Why R?

R is extremely powerful due to its flexibility. Furthermore if you stick to the build-in functions, R is also quite fast. Lastly R is free!
Continuing on build-in functions, let me show how to calculate the mean of a some numbers in another programming language (Perl):

```
my @nums = (1,2,3,4,5);
my $n    = scalar(@nums);
my $sum  = 0;
for(my $i=0;$i<=$#nums;$i++){
    $sum += $nums[$i];
}
my $mean = $sum / $n;
print $mean, "\n";
```

Now, let us try to do the same thing in R:

```
nums = c(1,2,3,4,5)
mean(nums)

## [1] 3
```

I hope you appreciate the difference!
NOTE! Since R does not come with a point-and-click interface, it is VERY useful to know, that the help page for any function is available by simply typing e.g. `?mean`

2 Variables

In this section, we will take a close look at what a variable is.
We saw earlier, that we can perform simple calculations in R, e.g.:

```
2 + 3

## [1] 5
```

However we will seldom write e.g. $2 + 3$, when we write R-code. More likely we will have the values 2 and 3 stored in appropriate 'containers'. This storing of 'a value' in 'a container' is in essense what a variable is, e.g.:

```
a = 2
b = 3
```

Here we *assign* the value 2 to a and the value 3 to b. Now we can perform the same calculations as before:

```
a + b

## [1] 5
```

As you see, we get the same results as before, only now, we are performing calculations using the variables a and b instead of the actual numbers 2 and 3.
The result of a calculation on two variables, can be assigned to a third variable like so:

```
c = a + b
c

## [1] 5
```

In the example above, the variables contain numbers. Variables can contain other things than numbers, i.e. vatiables will be of a certain data type:

1. numerical
2. character
3. logical

In the following, we will take a closer look at each of these data types.

2.1 Numeric variables

A numeric variable holds a number, e.g. as we saw earlier:

```
a = 2
b = 3
```

Familiar arithmetic variable operations

We can perform a range of arithmetic variable operations, such as the familiar:

```
# Plus
a + b

## [1] 5

# Minus
a - b

## [1] -1

# Multiply
a * b

## [1] 6

# Divide
a / b

## [1] 0.6666667
```

Special arithmetic variable operations

Furthermore, the following more special variable operations can be performed:

```
# Power
b ^ a

## [1] 9

# Power (same as above)
b ** a

## [1] 9
```

```r
# Square root
sqrt(a)
```

```
## [1] 1.414214
```

```r
# Which as you know really is
a ** (1/2)
```

```
## [1] 1.414214
```

```r
# Modulus (What is the remainder? E.g. 21 %% 4 = 1)
b %% a
```

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

```r
# Integer division (How many times is number wholly divisible? E.g. 21 %% 4 = 5)
b %/% a
```

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

Numbers in programming terminology

In programming terminology, numbers exist in different formats:

```r
# An integer is a whole number, e.g.:
3
```

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

```r
# A float is a decimal number, e.g.:
3.14159
```

```
## [1] 3.14159
```

```r
# A scientific number, is a number stated using scientific notation, e.g.:
6.63e-34
```

```
## [1] 6.63e-34
```

```r
# This is the same as
6.63 * 10 ^ -34
```

```
## [1] 6.63e-34
```

2.2 Character variables

A character variables hold a letter like so:

```r
a = "a"
b = "b"
a
```

```
## [1] "a"
```

```r
b
```

```
## [1] "b"
```

If we have multiple characters following each other we call it a string:

```
a = "This"
b = "is a string"
a
```

```
## [1] "This"
```

```
b
```

```
## [1] "is a string"
```

There are a plethora of string operators available. Usually we do not do much string manipulation in R. For now, we will stick to string concatenation (joining strings together in an end-to-end manner) and getting length of string (how many letters are in the string):

```
paste(a,b)
```

```
## [1] "This is a string"
```

```
nchar(b)
```

```
## [1] 11
```

2.3 Logic variables

This is a bit more tricky than the numerical and character data types. A logical is a representation of something which is either *TRUE* (same as 1) or *FALSE* (same as 0). Hence a logical variable is a *binary* variable with only TWO possible values.
We assign a value to a logical variable like so:

```
a = TRUE
b = FALSE
```

Note the absence of quotation marks and uppercase. We can perform the following logical (also known as boolean) operations:

```
# AND: TRUE AND FALSE =
a && b
```

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

```
# OR: TRUE OR FALSE =
a || b
```

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

```
# NOT: NOT TRUE =
!a
```

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

```
# Combining
!( (a && b) || (a || b) )
```

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

```
# Explanation
# !( (TRUE && FALSE) || (TRUE || FALSE) ) =
# !( FALSE || TRUE ) =
# !TRUE =
# FALSE
```

Note that *TRUE* and 1 are equivalent and likewise for *FALSE* and 0, e.g.:

```
a = 1
b = 0
a == TRUE

## [1] TRUE

b == FALSE

## [1] TRUE

!( (a && b) || (a || b) )

## [1] FALSE
```

These logicals/boolean values are used for comparing entities:

```
# Is 2 equal to 3?
2 == 3

## [1] FALSE

# Is 2 NOT equal to 3?
2 != 3

## [1] TRUE

# Is "a" equal to "b"?
"a" == "b"

## [1] FALSE

# Is "a" NOT equal to "b"?
"a" != "b"

## [1] TRUE

# Or test if sqrt(2) really is 2 ^ (1/2) as we stated earlier
sqrt(2) == 2 ^ (1/2)

## [1] TRUE
```

Here is a tabular representation of the above mentioned functions:

bool	bool	AND	OR
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

and here is a table of the comparators:

symbol	meaning
==	equal to
!=	not equal to
>	larger than
>=	larger than or equal to
<	smaller than
<=	smaller than or equal to

2.4 Special variable values

The concept of nothing/empty is actually quite tricky to handle in programming. In the following I will introduce the various values R use for representing nothing/empty and alike

The `NA` value

In R we also need a way to represent the absence of a value. Consider the case, where we have a series of incomplete measurements, which value do we assign the missing values? In R we represent missing values with `NA` (Not Available):

```
a = NA
a

## [1] NA

a == a

## [1] NA

is.na(a)

## [1] TRUE
```

Note how you cannot perform operations on the `NA` value, but you can ask R if the value is `NA`.

The `NaN` value

We also need a value for something which in numerical terms are not a number, for this we use `NaN` (Not a Number):

```
b = NaN
b

## [1] NaN

b == b

## [1] NA

is.nan(b)

## [1] TRUE

0 / 0

## [1] NaN
```

Note how you cannot perform operations on the `NaN` value, but you can ask R if the value is `NaN`.

The `inf/-inf` value

We also need to represent un-imaginable large numbers:

```
1 / 0

## [1] Inf

-1 / 0

## [1] -Inf
```

An likewise for un-imaginable small numbers

```
1 / 1e1000

## [1] 0
```

Note how this technically is not true, since `1 / 1e1000 == 1e-1000`, which R actually knows:

```
1 / 1e1000 == 1e-1000

## [1] TRUE
```

However for all intend and purpose, I hope we can agree that `1 / 1e1000 == 0`, which again R actually knows:

```
1 / 1e1000 == 0

## [1] TRUE
```

The `NULL` value

There is one more more special value we need to consider, namely the value of an empty variable, e.g.:

```
a = c()
b = NULL
is.null(a)

## [1] TRUE

is.null(b)

## [1] TRUE

a == b

## logical(0)
```

Again we cannot perform operations, but we can ask if the value is `NULL`.

2.5 Read-and-understand questions

1. What data type does the a variable with value `3` have?
2. What data type does the a variable with value `"a"` have?
3. What data type does the a variable with value `TRUE` have?
4. What is the difference between `"a"` and `a`?
5. What is the difference between `a = "a"` and `a == "a"`?
6. What is the result of `"a" == a`?
7. What is the result of `1 == TRUE`?
8. What is the result of `(TRUE && FALSE) || (FALSE || TRUE)`?
9. What is the result of `3 / 0`?
10. What is the result of `sqrt(-3)`?
11. What is the result of `2 ** 2 ^ 2`
12. What is the result of `32 %% 6`?
13. What is the result of `32 %/% 6`?
14. Is `a = ""` a `NULL` value? Why/why not?
15. Is `b = ""`, then what is the result of `nchar(b) == FALSE`?

3 Data containers

So far we have looked at what a variable is and what a data type of a variable is. Now we will look at how to store multipe values in one variable.

3.1 Vectors

A vector is a way to hold multiple values in one variable. Earlier you saw me defining a vector, when we looked at how to compute the mean of a set of numbers, like so:

```
nums = c(1,2,3,4,5)
nums

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

Vectors can also hold strings:

```
strings = c("this","is","a","string")
strings

## [1] "this"   "is"     "a"      "string"
```

or logical values:

```
logics = c(FALSE,FALSE,TRUE,TRUE,FALSE)
logics

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

However you CANNOT mix the values:

```
mixed = c(1,"a",FALSE)
mixed

## [1] "1"     "a"     "FALSE"
```

What happened here?
The individual number in this numerical vector is referred to as an element and can be accessed like so:

```
# Show me the FIRST element of strings:
strings[1]

## [1] "this"

# Change the 2nd element of strings to "constitutes":
strings[2] = "constitutes"
strings

## [1] "this"        "constitutes" "a"           "string"

# Show me the elements of strings at position 2,3,4
strings[2:4]

## [1] "constitutes" "a"           "string"

# Show me the elements of strings in reverse order
strings[4:1]

## [1] "string"      "a"           "constitutes" "this"
```

```
# Do Yoda!
paste(strings[c(3,4,1,2)],collapse=" ")

## [1] "a string this constitutes"

# Note the usage of a numerical vector in indexing the character vector
# and the wrapping concatenation function
```

IMPORTANT! Note how R is NOT zero-indexed! The first position in a vector is position 1!
Earlier you saw me calculating the mean of a set of numbers stored in a numerical vector using the function mean. Functions are what lies at the heart of R. In the following I will give examples of vector relevant functions:

```
# How many elements are in the numerical vector nums?
length(nums)

## [1] 5

# What is the reciprocal value of each of the numbers in the numerical vector nums?
1/nums

## [1] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000

# Give me a vector with all the numbers from 1 to 10
seq(1,10)

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

# Add the number 11 to the end of the above sequence of numbers
v = seq(1,10)
v

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

v = append(v,11)
v

##  [1]  1  2  3  4  5  6  7  8  9 10 11

# Remove the number 5
v = v[-5]
v

##  [1]  1  2  3  4  6  7  8  9 10 11

# Show me the first element
v[1]

## [1] 1

# Show me the last element
v[length(v)]

## [1] 11
```

Note above how some functions will work on all the elements and return one value, whereas others will work on the individual elements and return the resulting values in a new vector.
Just to give you a flavour for some basic descriptive statistics:

```
# What is the mean of the values in the numerical vector nums?
mean(nums)
```

```
## [1] 3

# What is the standard deviation of the values in the numerical vector nums?
sd(nums)

## [1] 1.581139

# What is the median of the values in the numerical vector nums?
median(nums)

## [1] 3
```

3.2 Matrices

A matrix is simply a table with some values in it. If we take a bunch of vectors, with the same length and put them on top of each other, we have a matrix. Think of an Excel sheet with a table, with m rows and n columns. These are VERY useful both for storing values and doing calculations on.
In R we can get a matrix A with $m = 3$ rows and $n = 5$ columns.

```
# Define the number of rows
m = 3
# Define the number of columns
n = 5
# Define the elements of the matrix
elements = seq(1:(m*n))
# View the elements of the matrix
elements

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

# Build matrix
A = matrix(data=elements,nrow=m,ncol=n)
# View matrix
A

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

Note had I stated `elements = 0`, then the value `0` would have been recycled, such that the value of ALL the elements in A would be `0`.
Like the vector, matrices can hold a single data type, i.e. either numbers, characters or logicals, but NOT a mix!
We can access a single element, e.g. the element in the 3rd row in the 2nd column, in the matrix like so:

```
A[3,2]

## [1] 6
```

Or ask for the content of the 3rd row

```
A[3,]

## [1]  3  6  9 12 15
```

Or the content of the 2nd column

```
A[,2]

## [1] 4 5 6
```

You can use negative indices to remove either a row or a column:

```
A[,-2]

##      [,1] [,2] [,3] [,4]
## [1,]    1    7   10   13
## [2,]    2    8   11   14
## [3,]    3    9   12   15
```

Think of e.g. 3 patients, where we have measured 5 clinical variables. Instead of using the indices, we can name the rows and columns like so:

```
rownames(A) = c("pat1","pat2","pat3")
colnames(A) = c("height","weight","HbA1c","bp_sys","bp_dia")
A

##      height weight HbA1c bp_sys bp_dia
## pat1      1      4     7     10     13
## pat2      2      5     8     11     14
## pat3      3      6     9     12     15
```

Now we can ask for values using the assigned names, e.g. what is the systolic blood pressure of pat2?

```
A["pat2","bp_sys"]

## [1] 11
```

If we want to know the dimensions of a given matrix, i.e. the number of rows and columns, we simply type

```
dim(A)

## [1] 3 5

nrow(A)

## [1] 3

ncol(A)

## [1] 5
```

Remember that the rows are ALWAYS indiced before the columns! (In Danish 'R' preceeds 'S' in the alphabet like so 'R'aekker and 'S'oejler).

3.3 Read-and-understand questions

1. What is the difference between `nchar("string")` and `length("string")`?
2. if `a = c("1","2","3")`, what type of vector is `a` then?
3. if `b = c(1,2,3)`, what type of vector is `b` then?
4. if `c = c(1,0,1)`, what type of vector is `c` then?
5. if `d = c(TRUE,FALSE,TRUE)`, what type of vector is `d` then?
6. if `e = c("TRUE","FALSE","TRUE")`, what type of vector is `e` then?

7. What is the result of `length(a)`?
8. What is the result of `length(append(a,c(4,5)))`?
9. if `A = matrix(data=0,nrow=2,ncol=5)`, then how many elements are in `A`? What are the dimensions of `A`?
10. What is the result of `dim(A)[1]`?
11. What is the result of `dim(A[,-3])`
12. What is the result of `means(seq(1,9))`?
13. What is the command for generating a matrix with 10 rows and 10 columns, containing only `NA` values?
14. In what situation could a matrix of `NA` values be useful?
15. What is the dimension of: a point? a line? a square? a cube?

3.4 That's all folks!

That is it for now. In the next workshop, we will take a closer look at how to load data matrices and perform simple statistical analyses.