Chapter 2: Variables

Section 2.1: Variables, data structures and basic Operations

In R, data objects are manipulated using named data structures. The names of the objects might be called "variables" although that term does not have a specific meaning in the official R documentation. R names are *case sensitive* and may contain alphanumeric characters(`a-z`,`A-z`,`0-9`), the dot/period(`.`) and underscore(_). To create names for the data structures, we have to follow the following rules:

- Names that start with a digit or an underscore (e.g. `1a`), or names that are valid numerical expressions (e.g. `.11`), or names with dashes ('-') or spaces can only be used when they are quoted: `` `1a` `` and `` `.11` ``. The names will be printed with backticks:

```
list( '.11' ="a")
#$`.11`
#[1] "a"
```

- All other combinations of alphanumeric characters, dots and underscores can be used freely, where reference with or without backticks points to the same object.
- Names that begin with `.` are considered system names and are not always visible using the `ls()`-function.

There is no restriction on the number of characters in a variable name.

Some examples of valid object names are: `foobar`, `foo.bar`, `foo_bar`, `.foobar`

In R, variables are assigned values using the infix-assignment operator `<-`. The operator `=` can also be used for assigning values to variables, however its proper use is for associating values with parameter names in function calls. Note that omitting spaces around operators may create confusion for users. The expression `a<-1` is parsed as assignment (`a <- 1`) rather than as a logical comparison (`a < -1`).

```
> foo <- 42
> fooEquals = 43
```

So `foo` is assigned the value of `42`. Typing `foo` within the console will output `42`, while typing `fooEquals` will output `43`.

```
> foo
[1] 42
> fooEquals
[1] 43
```

The following command assigns a value to the variable named `x` and prints the value simultaneously:

```
> (x <- 5)
[1] 5
# actually two function calls: first one to `<-`; second one to the `()`-function
> is.function(`(`)
[1] TRUE  # Often used in R help page examples for its side-effect of printing.
```

It is also possible to make assignments to variables using `->`.

```
> 5 -> x
> x
```

```
[1] 5
>
```

Types of data structures

There are no scalar data types in R. Vectors of length-one act like scalars.

- Vectors: Atomic vectors must be sequence of same-class objects.: a sequence of numbers, or a sequence of logicals or a sequence of characters. `v <- c(2, 3, 7, 10)`, `v2 <- c("a", "b", "c")` are both vectors.
- Matrices: A matrix of numbers, logical or characters. `a <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), nrow = 4, ncol = 3, byrow = F)`. Like vectors, matrix must be made of same-class elements. To extract elements from a matrix rows and columns must be specified: `a[1,2]` returns `[1] 5` that is the element on the first row, second column.
- Lists: concatenation of different elements `mylist <- list (course = 'stat', date = '04/07/2009', num_isc = 7, num_cons = 6, num_mat = as.character(c(45020, 45679, 46789, 43126, 42345, 47568, 45674)), results = c(30, 19, 29, NA, 25, 26 ,27) )`. Extracting elements from a list can be done by name (if the list is named) or by index. In the given example `mylist$results` and `mylist[[6]]` obtains the same element. Warning: if you try `mylist[6]`, R won't give you an error, but it extract the result as a list. While `mylist[[6]][2]` is permitted (it gives you 19), `mylist[6][2]` gives you an error.
- data.frame: object with columns that are vectors of equal length, but (possibly) different types. They are not matrices. `exam <- data.frame(matr = as.character(c(45020, 45679, 46789, 43126, 42345, 47568, 45674)), res_S = c(30, 19, 29, NA, 25, 26, 27), res_O = c(3, 3, 1, NA, 3, 2, NA), res_TOT = c(30,22,30,NA,28,28,27))`. Columns can be read by name `exam$matr`, `exam[, 'matr']` or by index `exam[1]`, `exam[,1]`. Rows can also be read by name `exam['rowname', ]` or index `exam[1,]`. Dataframes are actually just lists with a particular structure (rownames-attribute and equal length components)

Common operations and some cautionary advice

Default operations are done element by element. See `?Syntax` for the rules of operator precedence. Most operators (and may other functions in base R) have recycling rules that allow arguments of unequal length. Given these objects:

Example objects

```
> a <- 1
> b <- 2
> c <- c(2,3,4)
> d <- c(10,10,10)
> e <- c(1,2,3,4)
> f <- 1:6
> W <- cbind(1:4,5:8,9:12)
> Z <- rbind(rep(0,3),1:3,rep(10,3),c(4,7,1))
```

Some vector operations

```
> a+b # scalar + scalar
[1] 3
> c+d # vector + vector
[1] 12 13 14
> a*b # scalar * scalar
[1] 2
> c*d # vector * vector (componentwise!)
[1] 20 30 40
> c+a # vector + scalar
[1] 3 4 5
> c^2 #
[1]  4  9 16
> exp(c)
```

```
[1]  7.389056 20.085537 54.598150
```

Some vector operation Warnings!

```
> c+e # warning but.. no errors, since recycling is assumed to be desired.
[1] 3 5 7 6
Warning message:
In c + e : longer object length is not a multiple of shorter object length
```

R sums what it can and then reuses the shorter vector to fill in the blanks... The warning was given only because the two vectors have lengths that are not exactly multiples. c+f # no warning whatsoever.

Some Matrix operations Warning!

```
> Z+W # matrix + matrix #(componentwise)
> Z*W # matrix* matrix#(Standard product is always componentwise)
```

To use a matrix multiply: V %*% W

```
> W + a # matrix+ scalar is still componentwise
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
[4,]    5    9   13

> W + c # matrix + vector... : no warnings and R does the operation in a column-wise manner
     [,1] [,2] [,3]
[1,]    3    8   13
[2,]    5   10   12
[3,]    7    9   14
[4,]    6   11   16
```

"Private" variables

A leading dot in a name of a variable or function in R is commonly used to denote that the variable or function is meant to be hidden.

So, declaring the following variables

```
> foo <- 'foo'
> .foo <- 'bar'
```

And then using the `ls` function to list objects will only show the first object.

```
> ls()
[1] "foo"
```

However, passing `all.names = TRUE` to the function will show the 'private' variable

```
> ls(all.names = TRUE)
[1] ".foo"          "foo"
```