[Lab 1] A gentle introduction to R

Jae Yun JUN KIM,*

September 10, 2018

1 Installation of RStudio

Open a web browser and go to the RStudio site download site:
https://www.rstudio.com/products/rstudio/download/.
In this site, search for the free version of the `RStudio Desktop` and click the link `here` to download the `R`. Then, search for the section of `Installers for Supported Platforms`. Afterwards, download and install the `RStudio installer` that suits the most to your machine and to your operating system. Once the `RStudio` is installed, open it and you should see an IDE similar to the one shown below:

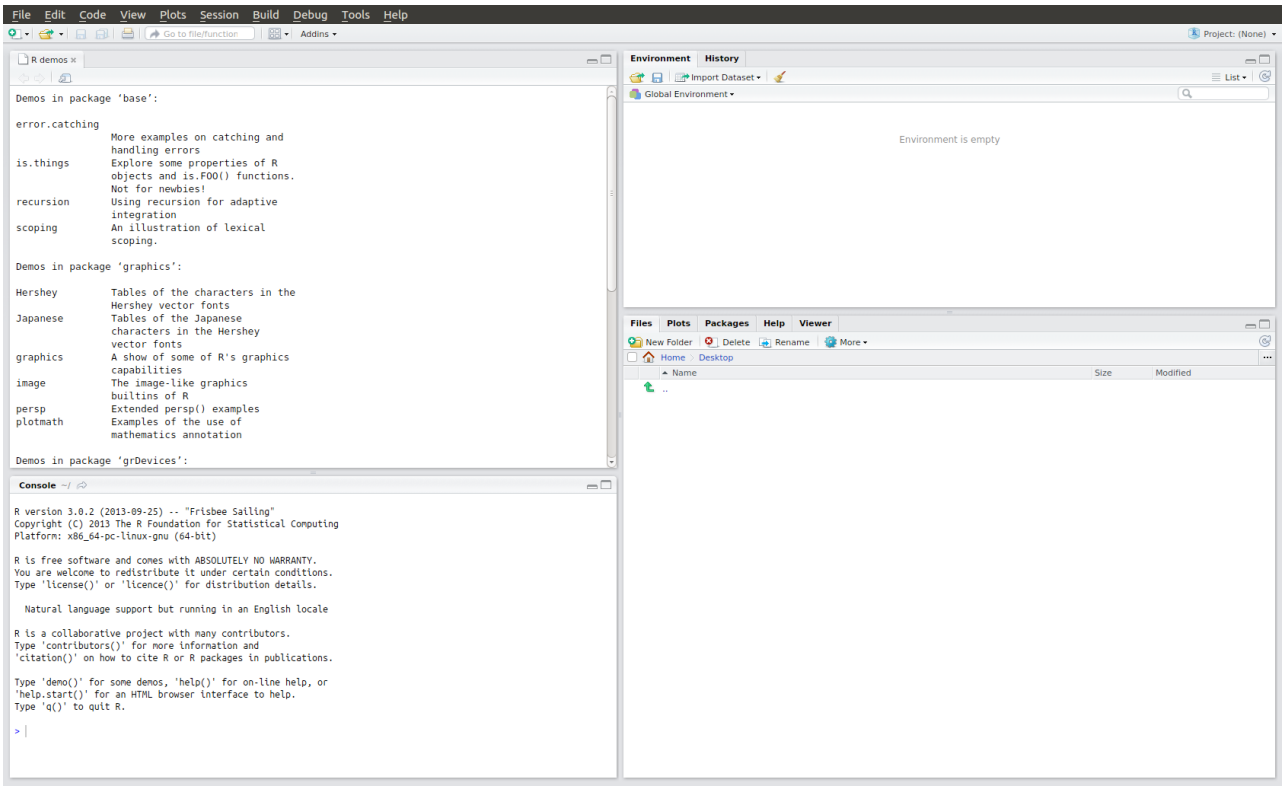

Figure 1: RStudio IDE

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle CS71520 75 725 Paris 15, France; jaeyun.jun-kim@ece.fr

2 Simple manipulations: numbers and vectors

2.1 Vectors and assignment

The assignment operator commonly used is ('<−'), which points to the object receiving the value of the expression. In most contexts the operator '=' can be used as an alternative. In this tutorial, for simplicity the operator '=' will be used.
Type the following statements to create vectors and to assign the resulting vector to a variable named `x`:

```
x = c(10.4, 5.6, 3.1, 6.4, 21.7)
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
assign ("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

2.2 Regular sequences

R has a number of facilities for generating commonly used sequences of numbers. Type the following expressions and see their effects:

```
a = 1:30
b = 2*1:15
```

A backward sequence can be generated as

```
c = 30:1
```

Alternatively, the function `seq`() can be used to generate sequences in a more general manner. It has five arguments and only some of which may be specified in any one call. For instance, type the following expression

```
seq(2,10)
```

which corresponds to $2 : 10$.

Try also the following expression

```
seq(from=-5,to=5,by=.2)
```

which corresponds to $\mathtt{c}(-5.0, -4.8, -4.6, \cdots, 4.6, 4.8, 5.0)$.

The same results can be achieved by the following expression

```
s4 = seq(length=51,from=-5,by=.2)
```

Another argument type for the `seq`() function is

```
seq(along=s4)
```

which will generate a vector like `c(1, 2, 3, ···, length(s4))`.

Finally, the function `rep()` can be used for replicating an object in various complicated ways. Type the following expression

```
s5 = rep(x, times=5)
```

which will copy `x` end-to-end and assign the resulting vector to `s5`.

On the other hand,

```
s6 = rep(x, each=5)
```

repeats each element of `x` five times before moving on to the next.

2.3 Logical vectors

Logical vectors are generated by conditions, and the elements of a logical vector can be of: `TRUE`, `FALSE`, and `NA` (i.e., Not-Available).

Type the following expression

```
temp = x > 13
```

which sets `temp` as a vector of the same length as `x`, with values `FALSE` corresponding to elements of `x` where the condition is not met and `TRUE` where it is.

The logical operators that can be used for generating logical vectors are: $<, <=, >, >=, ==, != $.

In addition, if `c1` and `c2` are some logical expressions, then `c1 & c2` is their intersection ("and"), `c1 | c2` is their union ("or"), and `!c1` is the negation of `c1`.

2.4 Missing values

In some cases the components of a vector may not be completely known. When an element is "not available", a place within a vector may be reserved for it by assigning it the special value `NA`.

Type the following expression:

```
z = c(-1:3,NA)
```

The function `is.na(z)` gives a logical vector of the same size as `z` with value `TRUE` if and only if the corresponding element in `z` is `NA`.

Type the following expression:

```
ind = is.na(z)
```

Notice that the logical expression `z == NA` is quite different from `is.na(z)`, since NA is not really a value but a marker for a quantity that is not available. Thus, `x == NA` is a vector of the same length as `z`, all of whose values are `NA`, as the logical expression itself is incomplete and undecidable.
Note that there is a second kind of "missing" values which are produced by numerical computation, the so-called **Not a Number** (`NaN`).

Type the following expressions:

```
0/0
```

or

```
Inf - Inf
```

which both give `NaN`, since the result cannot be defined sensibly.

2.5 Character vectors

A character vector is a sequence of characters delimited by the double quote character. For example,

```
"x-values"
"New iteration results"
```

They use C-style escape sequences using \ as the escape character. Some useful escape sequences are \n, for newline, \t, for tab, and \b, for backspace. In addition, \\ is entered and printed as \. Inside double quotes " is entered as \".
See ?`Quotes` for a full list.
Character vectors may be concatenated into a vector by the `c`() function examples.
The `paste`() function takes an arbitrary number of arguments and concatenates them one by one into character strings. The arguments are by default separated in the result by a single blank character, but this can be changed by the named argument `sep = string`.
Type the following expression:

```
labs = paste(c("X","Y"),1:10,sep="")
```

which is equivalent to `labs`=`c`("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10").
Note that `c`("X", "Y") is repeated 5 times to match the sequence `1 : 10`.

2.6 Index vectors: selecting and modifying subsets of a data set

Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets.
Such index vectors can be any of four distinct types.

2.6.1 Logical vector

In this case, the index vector is recycled to the same length as the vector from which elements are to be selected. Values corresponding to `TRUE` in the index vector are selected and those corresponding to `FALSE` are omitted.

Type the following expression:

```
y = z[!is.na(z)]
```

Try also

```
(z+1)[(!is.na(z)) & z > 0]
```

which creates an object `z` and places in it the values of the vector $\texttt{z} + 1$ for which the corresponding value in `z` was both non-missing and positive.

2.6.2 A vector of positive integral quantities

In this case, the values in the index vector must lie in the set $\{1, 2, \cdots, \texttt{length(x)}\}$.

Type the following expression:

```
x[1:10]
```

which selects the first 10 elements of `x` (assuming that `length(x)` is not less than 10).

Try also

```
c("x","y")[rep(c(1,2,2,1), times=4)]
```

which produces a character vector of length 16 consisting of "`x`", "`y`", "`y`", "`x`" repeated four times.

2.6.3 A vector of negative integral quantities

Such an index vector specifies the values to be excluded rather than included.

Type the following expression:

```
y = x[-(1:5)]
```

which gives `y` all but the first five elements of `x`.

2.6.4 A vector of character strings

This possibility only applies where an object has a **names** attribute to identify its components. In this case, a sub-vector of the names vector may be used in the same way as the positive integral labels.

Type the following expressions:

```
fruit = c(5,10,1,20)
names(fruit) = c("orange", "banana", "apple", "peach")
lunch = fruit[c("apple","orange")]
```

The advantage is that alphanumeric *names* are often easier to remember than *numeric indices*.

Notice that an indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed *only on those elements of the vector*.

Type the following expression

```
z[is.na(z)] = 0
```

which replaces any missing values in `z` by zeros.

Try also,

```
z[z < 0] = -z[z < 0]
```

which has the same effect as `z = abs(z)`.

3 Arrays and matrices

3.1 Index matrices

An array can be considered as a multiply subscripted collection of data entries. A dimension vector is a vector of non-negative integers.

Type the following expression to create a 4 by 5 array `x`

```
x = array(1:20, dim=c(4,5))
```

We now wish to replace the values of $\texttt{x}[1,3], \texttt{x}[2,2]$, and $\texttt{x}[3,1]$ by zero value. To achieve this goal, one can do

```
i = array(c(1:3,3:1), dim=c(3,2))
x[i] = 0
```

Negative indices are not allowed in index matrices.

`NA` and zero values are allowed: rows in the index matrix containing a zero are ignored, and rows containing an `NA` produce an `NA` in the result.

To combine vector, matrix or data frame by columns, the command `cbind`() can be used. On the other hand, to combine vector, matrix or data frame by rows, the command `rbind`() can be used.

Type the following expressions

```
x1 = read.csv("data1.csv",header=T, sep=",")
x2 = read.csv("data2.csv",header=T, sep=",")
x3 = cbind(x1,x2)
x3
x4 = rbind(x1,x1)
```

```
x4
```

If you get an error message that says ‘‘Error in file(file, ‘‘rt") : cannot open the connection", make sure that the above files are located in your current working directory. First, you can find out your current working directory by typing

```
getwd()
```

And you can set a new working directory by typing

```
setwd('Your_new_working_directory')
```

where you need replace Your_new_working_directory by your desired working directory. If your operating system is Microsoft Windows, then make sure that the symbol used to separate the folders is / (and not \).

3.2 The array() function

Arrays can be constructed from vectors by the **array** function:
z = array(< *data_vector* >, < *dim_vector* >

Type the following expression:

```
h = 1:24
z = array(h, dim=c(3,4,2))
```

The above expressions define a sequence h with 24 numbers, and reshapes it as an 3 by 4 by 2 array for z.

The same results can be achieved with the following expressions. Try them

```
z = h
dim(z) = (3,4,2)
```

Another example is

```
z = array(0, c(3,4,2))
```

which makes z an array of all zeros.

Arrays may be used in arithmetic expressions and the result is an array formed by element-by-element operations on the data vector.

Type the following expressions

```
a = c(1,2,3)
b = c(2,4,6)
c = cbind(a,b)
x = c(2,2,2)
c
x
```

In R the asterisk (*) is used for element-wise multiplication. This is where the elements in the same row are multiplied by one another.

Type the following operations which will give the same results

```
c*x
x*c
```

3.3 The outer product of two arrays

An important operation on arrays is the **outer product**. If `a` and `b` are two numeric arrays, their outer product is an array whose dimension vector is obtained by concatenating their two dimension vectors (order is important). The outer product is formed by the special operator %o%.
Type the following expression:

```
a %o% b
```

An alternative way to achieve the same goal is

```
ab = outer(a,b,"*")
```

Furthermore, the multiplication function can be replaced by an arbitrary function of two variables. For example if we wish to evaluate the function $\mathtt{f}(\mathtt{x},\mathtt{y}) = \cos(\mathtt{y})/(1+\mathtt{x}^2)$ over a regular grid of values of $x-$ and $y-$ coordinates.

Try the following expressions:

```
f = function(x,y) cos(y) / (1+x^2)
z = outer(a,b,f)
```

In particular, the outer product of two ordinary vectors is a doubly subscripted array. Notice that the outer product operator is non-commutative.

3.4 Generalized transpose of an array

The function `aperm(a, perm)` may be used to permute an array `a`. The argument `perm` must be a permutation of the integers $\{1, \cdots, k\}$, where `k` is the number of subscripts in `a`. The result of the function is an array of the same size as `a` but with old dimension given by perm[j] becoming the new `j`-th dimension. The easiest way to think of this operation is as a generalization of transposition for matrices. Indeed, if `c` is a matrix, then `d` given by

```
d = aperm(c,c(2,1))
```

is just the transpose of c. For this special case, a simpler function `t()` is available. Hence,

```
e = t(c)
```

also transposes the matrix `c`.

3.5 Matrix facilities

3.5.1 Matrix definition

A matrix is a collection of data elements arranged in a two-dimensional layout. Suppose that we have a matrix with 2 rows and 3 columns:

$$A = \begin{bmatrix} 3 & 1 & 5 \\ 9 & 10 & 4 \end{bmatrix} \tag{1}$$

This matrix A can be defined in `R` as follows

```
A = matrix(c(3,1,5,9,10,4), nrow=2, ncol=3,byrow=TRUE)
```

One can achieve the same goal by doing

```
A = matrix(c(3,9,1,10,5,4), nrow=2, ncol=3,byrow=FALSE)
```

One can access the `i`$^{\texttt{th}}$ row and `j`$^{\texttt{th}}$ column using the expression `A[i,j]`. For example,

```
A[1,3]
```

The entire `i`$^{\texttt{th}}$ row can be extracted as `A[i,]`. Similarly the entire `j`$^{\texttt{th}}$ column can be extracted as `A[,j]`. Type the following expressions

```
A[2,]
A[,2]
```

We can also extract more than one row or column at a time. For example,

```
A[,c(1,3)]
```

3.5.2 Matrix multiplication

The operator $\%*\%$ is used for matrix multiplication, while the operator $*$ is used for element-by-element products.
Define matrices `B` and `C` with the same size (e.g. 3 by 3) typeing the following expressions

```
B = matrix(c(1,2,3,6,5,4,7,9,8),nrow=3, ncol=3, byrow=TRUE)
C = matrix(9:1,nrow=3, ncol=3, byrow=TRUE)
B*C
```

to perform element-by-element products, and

```
B %*% C
```

for a matrix product.

In addition, define a 3 by 1 vector `x` and type the following quadratic-form expression

```
x = c(2,2,2)
x %*% B %*% x
```

Notice that vectors which occur in matrix multiplication expressions are automatically promoted either to row or column vectors, whichever is multiplicatively coherent, if possible.

The function `crossprod()` forms "crossproducts", meaning that `crossprod(x, y)` is the same as `t(x) %*% y` but the former command is more efficient. If the second argument to `crossprod()` is omitted, then the command takes it to be the same as the first one. Try the following expressions

```
y=c(1,2,3)
crossprod(x,y)
crossprod(y)
```

The functions `nrow(A)` and `ncol(A)` give the number of rows and columns of the matrix `A`, respectively. Try

```
nrow(A)
ncol(A)
```

The meaning of `diag()` depends on its argument. For example, `diag(y)`, where `y` is a vector, gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand, `diag(C)`, where `C` is a matrix, gives the vector of main diagonal entries of `C`. Finally, if `k` is a positive scalar, then `diag(k)`is the `k` by `k` identity matrix. Try the following expressions

```
diag(y)
diag(C)
diag(4)
```

3.5.3 Linear equations and inversion

Consider a linear problem of $\mathtt{Bx} = \mathtt{b}$, where `B` and `b` are known but `x` are unknown. Then, one can solve them by

```
x = solve(B,b)
```

The mathematical expression for `x` is $\mathtt{x} = \mathtt{B}^{-1}\mathtt{b}$, where $\mathtt{B}^{-1}$ denotes the inverse of `B`, which can be computed by `solve(B)`.
On the other hand, the quadratic form $\mathtt{x}^{\mathtt{T}}\mathtt{B}^{-1}\mathtt{x}$ should be computed as

```
x %*% solve(B,x)
```

rather than computing `x %*% solve(A) %*% x`, which can be very inefficient.

3.5.4 Eigenvalues and eigenvectors

The function `eigen(Sm)` calculates the eigenvalues and eigenvectors of a symmetric matrix `Sm`. The result of this function is a list of two components named `values` and `vectors`. The assignment `ev = eigen(Sm)` will assign this list to `ev`. Then `ev$val` is the vector of eigenvalues of `Sm` and `ev$vec` is the matrix of corresponding eigenvectors. Had we only needed the eigenvalues, we could have used the assignment `evals = eigen(Sm)$values`. Try the following expressions

```
Sm = matrix(c(1,7,3,7,4,-5,3,-5,6), nrow=3, ncol=3, byrow=TRUE)
ev = eigen(Sm)
ev$val
ev$vec
evals = eigen(Sm)$values
```

3.5.5 Singular value decomposition and determinants

The function `svd(B)` takes an arbitrary matrix argument `B` and calculates the singular value decomposition of `B`. This consists of a matrix of orthonormal columns `U` with the same column space as `B`, a second matrix of orthonormal columns `V`, whose column space is the row space of `B`, and a diagonal matrix of positive entries `D` such that `B = U %*% D %*% t(V)`. `D` is actually returned as a vector of the diagonal elements. Hence, the result of `svd(B)` is actually a list of three components named `d`, `u` and `v`. Try

```
svdB = svd(B)
svdB$d
svdB$u
svdB$v
```

If `B` is square, then, one can calculate the absolute value of the determinant of `B` as

```
absDetB = prod(svd(B)$d)
```

One can define a function that calculates the absolute value of the determinant of a matrix as

```
absDet = function(A) prod(svd(A)$d)
```

Then, we can use this function to compute the absolute value of the determinant of the matrix `B` as

```
absDet(B)
```

We can also define function to compute the trace of a square matrix as

```
trace = function(A) sum(diag(A))
```

Then, we can use this function as follows

```
trace(B)
```

One can also use the built-in function `det`() to calculate a determinant including the sign, and `determinant` to give the sign and modulus of the determinant of a given matrix. Try

```
det(B)
determinant(B)
```

3.6 Forming partitioned matrices with `cbind()` and `rbind()`

`cbind`() forms matrices by binding together matrices horizontally (or column-wise), and `rbind`() vertically (or row-wise). For `X` = `cbind`(*arg_1*, *arg_2*, *arg_3*, $\cdots$) the arguments to `cbind`() must be either vectors of any length, or matrices with the same column size, that is, the same number of rows. The result is a matrix with the concatenated arguments `arg_1`, `arg_2`, $\cdots$.

The function `rbind`() does the corresponding operation for rows. In this case, any vector argument, possibly cyclically extended, are taken as row vectors.

Try the following expressions

```
cbind(a,b)
rbind(a,b)
```

3.7 Concatenation function `c()` with arrays

It should be noted that whereas `cbind`() and `rbind`() are concatenation functions that respect `dim` attributes, the basic `c`() function does not, but rather clears numeric objects of all `dim` and `dimnames` attributes. Try

```
dim(cbind(a,b))
dimnames(cbind(a,b))
```

To coerce an array back to a simple vector object, one can do

```
vec = as.vector(cbind(a,b))
```

Or,

```
vec1 = c(cbind(a,b))
```

where the first method is preferable.

4 Lists and data frames

4.1 Lists

An `R list` is an object consisting of an ordered collection of objects known as its `components`.

There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. Below is given a simple example of a list:

```
Lst = list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))
```

The `components` are always *numbered* and may always be referred to as such. Thus, if `Lst` is the name of a list with four components, these may be individually referred to as:

```
Lst[[1]]
Lst[[2]]
Lst[[3]]
Lst[[4]]
```

Furthermore, if `Lst`[[4]] is a vector subscripted array, then

```
Lst[[4]][1]
Lst[[4]][2]
Lst[[4]][3]
```

The components of lists may also be *named* and can be accessed also by

```
Lst$name
Lst$wife
Lst$no.children
Lst$child.ages
Lst$child.ages[1]
Lst$child.ages[2]
Lst$child.ages[3]
```

Or by,

```
Lst[["name"]]
Lst[["wife"]]
Lst[["no.children"]]
Lst[["child.ages"]]
Lst[["child.ages"]][1]
Lst[["child.ages"]][2]
Lst[["child.ages"]][3]
```

4.2 Constructing, modifying, and concatenating lists

New lists may be formed from existing objects by the function `list`(). An assignment of the form `Lst` = `list`(*name_1* = *object_1*, $\cdots$, *name_m* = *object_m*) sets up a list `Lst` of m components using *object_1*, $\cdots$, *object_m* for the components and giving them names as specified by argument names (which can be freely chosen).

Lists can be extended by specifying additional components. For example,

```
Lst[5] = list(matrix=B)
```

Finally, one can concatenate various existing lists. For example,

```
Lst2 = list(vector=a)
Lst3 = c(Lst,Lst2)
```

4.3 Data frames

A `data frame` is a list with class "`data.frame`".
A data frame may be regarded as a matrix with columns, possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

4.3.1 Making data frames

Objects satisfying the restrictions placed on the columns (components) of a data frame may be used to form one using the function `data.frame`:

```
a = c(1,2,3)
b = c(2,4,6)
c = cbind(a,b)
df1 = data.frame(c)
df2 = data.frame(2*c)
df1
df2
```

In addition, a list whose components conform to the restrictions of a data frame may be *coerced* into a data frame using the function `as.data.frame`(). Try

```
as.data.frame(Lst3)
```

Finally, the simplest way to construct a data frame from scratch is to use the `read.table`() function to read an entire data frame from an external file. Try

```
z = read.table("data3.txt",header=T,sep="\t")
```

Make sure that your working directory contains the file to be read. `header=T` means that the first line in the file is a header. `sep="\t"` means that the columns are separated by tabs.

5 Grouping, loops and conditional execution

`R` is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used.

5.1 Conditional execution: if statements

One can write conditional statements in `R` as

`if`(*expr_1*) *expr_2*

`else` *expr_3*

Try

```
x = -5
if(x > 0){
  print("Nonnegative number")
}else{
  print("Negative number")
}
```

One can also write the conditional statements as

```
if(x > 0) print("Nonnegative number") else print("Negative number")
```

While the logic operators & and | apply element-wise to vectors, the operators && and || apply to vectors of length one and evaluate their second argument only if necessary.

There is a vectorized version of the `if`/`else` construct, the `ifelse` function. This has the form `ifelse`(`condition`, `a`, `b`) and returns a vector of the length of its longest argument, with elements `a`[`i`] if `condition`[`i`] is true, otherwise `b`[`i`]. Try

```
g = c(5,7,2,9)
ifelse(g %% 2 == 0, "even", "odd")
```

5.2 Repetition execution: for loops, repeat and while

First, loops can be implemented using the expression `for` as `for`(`name` `in` *expr_1*) *expr_2*, where `name` is the loop variable. *expr_1* is a vector expression, and *expr_2* is often a grouped expression with its sub-expressions written in terms of the variable `name`. *expr_2* is repeatedly evaluated as `name` ranges through the values in the vector result of *expr_1*. For example,

```
phi = -1
for (k in 1:100){
  phi = phi + 1
  print(phi)
}
```

Loops can also be implemented using the expression `while` as `while` (*condition*) *expr*. For example,

```
i = 1
while (i < 6){
  print(i)
  i = i+1
}
```

One can also use the expression `repeat` to implement a loop statement. For example,

```
x = 1
repeat{
  print(x)
  x = x+1
  if (x == 6){
    break
```

```
  }
}
```

Note that `break` statement can be used to terminate a loop. This is the only way to terminate `repeat` loops.

The `next` statement can be used to discontinue one particular cycle and skip to the "next".

6 Writing your own functions

A function is defined by an assignment of the form

$$\texttt{name} = \texttt{function}(\textit{arg\_1}, \textit{arg\_2}, \cdots) \quad \textit{expression}$$

where the *expression* is an R expression that uses the arguments *arg_i* to calculate a value. The value of the expression is the value returned for the function.

A call to the function then usually takes the form `name`(*expr_1*, *expr_2*, $\cdots$).

6.1 Simple examples

6.1.1 Calculation of the two sample t-statistic

Let us first define a function that calculates `two-sample t-statistic test` as follows

```
twosam = function(y1, y2) {
  n1  = length(y1); n2  = length(y2)
  yb1 = mean(y1);   yb2 = mean(y2)
  s1  = var(y1);    s2  = var(y2)

  s   = ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
  tst = (yb1-by2) / sqrt(s*(1/n1+1/n2))
  return(tst)
}
```

And one can use this function as

```
result = twosam(a,b)
result
```

6.1.2 MATLAB's backslash command

Design a function to emulate directly the MATLAB backslash command, which returns the coefficients of the orthogonal projection of the vector `y` onto the column space of the matrix `X`. This is ordinarily called the least squares estimate of the regression coefficients. This would ordinarily be done with the `qr`() function; however, this is sometimes a bit tricky to be used directly, and it pays to have a simple function such as the following.

Thus, given an `n` by `1` vector `y` and an `n` by `p` matrix `X`, `X\y` is defined as $(\texttt{X}^\texttt{T}\texttt{X})^\dagger\texttt{X}^\texttt{T}\texttt{y}$, where $(\texttt{X}^\texttt{T}\texttt{X})^\dagger$ is a generalized inverse of $\texttt{X}^\texttt{T}\texttt{X}$ and ^{T} is the transposition operation of a vector or a matrix. Try

```
bslash = function(X,y){
  X = qr(X)
  qr.coef(X,y)
}
```

After this object is created, it may be used in statements such as

```
regcoeff = bslash(B,y)
```

6.2 Named arguments and defaults

If arguments of a function are given in the "**name=object**" form, they may be given in any order. Furthermore, the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.
If a function `fun1` is defined as

$$\texttt{fun1} = \texttt{function}(\texttt{data}, \texttt{data.frame}, \texttt{graph}, \texttt{limit})\{[\textit{function body omitted}]\}$$

then the function may be invoked in several ways as

$$\begin{aligned}
&\texttt{ans} = \texttt{fun1}(\texttt{d}, \texttt{df}, \texttt{TRUE}, 20)\\
&\texttt{ans} = \texttt{fun1}(\texttt{d}, \texttt{df}, \texttt{graph} = \texttt{TRUE}, \texttt{limit} = 20)\\
&\texttt{ans} = \texttt{fun1}(\texttt{data} = \texttt{d}, \texttt{limit} = 20, \texttt{graph} = \texttt{TRUE}, \texttt{data.frame} = \texttt{df})
\end{aligned}$$

where all of them are equivalent.

In many cases, arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if `fun1` were defined as

$$\texttt{fun1} = \texttt{function}(\texttt{data}, \texttt{data.frame}, \texttt{graph} = \texttt{TRUE}, \texttt{limit} = 20)\{\cdots\}$$

Then, the function could be called as

$$\texttt{ans} = \texttt{fun1}(\texttt{d}, \texttt{df})$$

or as

$$\texttt{ans} = \texttt{fun1}(\texttt{d}, \texttt{df}, \texttt{limit} = 10)$$

which changes one of the defaults.
Note that defaults may be arbitrary expressions, even involving other arguments to the same function. They are not restricted to be constants.

6.3 Assignments within functions

Note that *any ordinary assignment done within a function are local and temporary and are lost after exiting from the function.* Thus, the assignment $\texttt{X} = \texttt{qr}(\texttt{X})$ does not affect the value of the argument in the calling program.
If global and permanent assignments are intended within a function, then either the "superassignment" operator, $<<-$, or the function `assign()` can be used.

7 Graphical procedures

7.1 Line plots

First, plots with default markers can be drawn as

```
stock1 = c(1,3,6,4,9)
plot(stock1)
```

One can now add a title to the figure and connect the markers using some color:

```
stock1 = c(1,3,6,4,9)
plot(stock1, type="o", col="blue")
title(main="Stocks", col.main="red", font.main=4)
```

We can also plot two curves simultaneously

```
stock1 = c(1,3,6,4,9)
stock2 = c(2,5,4,5,12)
plot(stock1, type="o", col="blue", ylim=c(0,12))
lines(stock2, type="o", pch=22, lty=2, col="red")
title(main="Stocks", col.main="red", font.main=4)
```

Now let us change the axes labels, add a legend, and compute the y-axis values using the `max` fucntion so any changes to our data will be automatically reflected in the graph

```
stock1 = c(1,3,6,4,9)
stock2 = c(2,5,4,5,12)
g_range = range(0,stock1,stock2)
plot(stock1, type="o", col="blue", ylim=g_range, axes=FALSE, ann=FALSE)
axis(1, at=1:5, lab=c("Mon","Tue","Wed","Thu","Fri"))
axis(2, las=1, at=4.0:g_range[2])
box()

lines(stock2, type="o", pch=22, lty=2, col="red")
title(main="Stocks", col.main="red", font.main=4)
title(xlab="Days", col.lab=rgb(0,0.5,0))
title(ylab="US $", col.lab=rgb(0,0.5,0))

legend(1, g_range[2], c("stock1","stock2"), cex=0.8,
       col=c("blue","red"), pch=21:22, lty=1:2)
```

7.2 Bar charts

Suppose that we have some stock-price data stored in a file. Draw the stock price corresponding to `stock1` using the `barplot`:

```
stock_data = read.table("data4.txt", header=T, sep="\t")
barplot(stock_data$stock1, main="Stock1", xlab="Days", ylab="US $",
        names.arg = c("Mon","Tue","Wed","Thu","Fri"), border="blue")
```

7.3 Histograms

Using the data read from the previous data file, we can also draw the histogram of the data as

```
stock_data = read.table("data4.txt", header=T, sep="\t")
stock = c(stock_data$stock1,stock_data$stock2,stock_data$stock3)
hist(stock,col="lightblue",ylim=c(0,10))
```

7.4 Pie charts

Draw a pie chart from the stock data

```
stock_data = read.table("data4.txt", header=T, sep="\t")
stock2 = stock_data$stock2
pie(stock2, main="stock2", col=rainbow(length(stock2)),
    labels=c("Mon","Tue","Wed","Thu","Fri"))
```

7.5 Displaying multivariate data

R provides two very useful functions for representing multivariate data. If `X` is a numeric matrix or data frame, the command

`pairs(X)`

produces a pairwise scatterplot matrix of the variables defined by the columns of `X`, that is, every column of `X` is plotted against every other column of `X`, and the resulting $\mathtt{n}(\mathtt{n}-1)$ plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

When three or four variables are involved a `coplot()` may be more enlightening. If `a` and `b` are numeric vectors, and `c` is a numeric vector or factor object (all of the same length), then the command

`coplot(a~b|c)`

produces a number of scatterplots of `a` against `b` for given values of `c`. If `c` is a factor, this simply means that `a` is plotted against `b` for every level of `c`. If `c` is a numeric vector, it is divided into a number of *conditioning intervals* and for each interval `a` is plotted against `b` for values of `c` within the interval. The number and position of intervals can be controlled with `given.values` = argument to `coplot()`.

7.6 Other display graphics

Some other display graphics used in R are the following. Consult the R documentation to get help on these display-graphics functions.
`qqnorm(x)`
`qqline(x)`

`qqplot(x, y)`

`hist(x)`
`hist(x, nclass = n)`
`hist(x, breaks = b, ···)`

`dotchart(x, ···)`
`image(x, y, z, ···)`
`contour(x, y, z, ···)`
`persp(x, y, z, ···)`

7.7 Low-level plotting commands

Sometimes the high-level plotting functions do not produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot. Some of the more useful low-level plotting functions are:

`points(x, y)`
`lines(x, y)`
`text(x, y, labels, ···)`
`abline(a, b)`
`abline(h = y)`
`abline(v = x)`
`abline(lm.obj)`
`polygon(x, y, ···)`
`legend(x, y, legend, ···)`
`title(main, sub)`
`axis(side, ···)`

7.7.1 Using graphics parameters

For permanent changes on the graphics parameters, the `par()` function can be used to access and modify the list of graphics parameters for the current graphics device.

`par()`

It returns a list of all graphics parameters and their values for the current device.

`par(c("col","lty"))`

It returns only the named graphics parameters.

`par(col = 4, lty = 2)`

It sets the values of the named graphics parameters and returns the original values of the parameters as a list.

8 Input/Output files

8.1 Save and load Data

An example to store some data into a `Rdata` file:

```
seq_a=1:10
save(seq_a, file="data_seq.Rdata")
```

Now delete the generated sequence and try to load the stored sequence as

```
rm(seq_a)
load("data_seq.Rdata")
print(seq_a)
```

8.2 Import and Export data to .csv Files

Do a similar task of writing but now working with .csv file:

```
var1 = 1:5
var2 = (1:5)/100
var3 = c("R", "and", "Data Mining", "Examples", "Case Studies")

df1 = data.frame(var1, var2, var3)

names(df1)= c("var_int", "var_real", "var_char")

write.csv(df1, file="var.csv", row.names=FALSE)
```

Similarly, read the data stored in the .cvs file:

```
df2= read.csv("var.csv")
```

8.3 Import and Export data to .txt Files

Do a similar task of writing but now working with .txt file:

```
var1 = 1:5
var2 = (1:5)/100
var3 = c("R", "and", "Data Mining", "Examples", "Case Studies")

df1=data.frame(var1, var2, var3)

names(df1)= c("var_int", "var_real", "var_char")

write.table(df1, file="var.txt", sep="\t", dec=".", row.names=FALSE)
```

Similarly, read the data stored in the .txt file:

```
df4 = read.table(file="var.txt", sep="\t", header=TRUE, dec=".")
```

9 Useful R commands

1. Use `getwd()` and `setwd("<desireddirectory>")` to get and set the console working directory.
2. To list objects: `ls()` or `objects()`.
3. To remove all the objects loaded in a R console: `rm(list = ls())`.
4. To remove only the functions (but not the variables) using: `rm(list = lsf.str())`.
5. To remove all objects except for functions: `rm(list = setdiff(ls(), lsf.str()))`.
6. To list available R packages: `library()`.
7. To list the databases attached to the system: `search()`.
8. To activate a package: `require(`*name_package*`)`.
9. To attach or detach a database: `attach(`*name_database*`)`, `detach(`*name_database*`)`.

10 Exercises

10.1 Objects and Arithmetic

1. Define

   $\texttt{x} = \texttt{c}(4, 2, 6)$

   $\texttt{y} = \texttt{c}(1, 0, -1)$

   Decide what the results will be of the following:

   (a) `length(x)`
   (b) `sum(x)`
   (c) `sum(x^2)`
   (d) $\texttt{x} + \texttt{y}$
   (e) $\texttt{x} * \texttt{y}$
   (f) $\texttt{x} - 2$
   (g) `x^2`

   Use R to check your answers.

2. Decide what the following sequences are and use R to check your answers:

   (a) $7 : 11$
   (b) $\texttt{seq}(2, 9)$
   (c) $\texttt{seq}(4, 10, \texttt{by} = 2)$
   (d) $\texttt{seq}(3, 30, \texttt{length} = 10)$

(e) `seq(6, −4, by = −2)`

3. Determine what the result will be of the following R expressions, and then use R to check that you are right:

   (a) `rep(2, 4)`
   (b) `rep(c(1, 2), 4)`
   (c) `rep(c(1, 2), c(4, 4))`
   (d) `rep(1 : 4, 4)`
   (e) `rep(1 : 4, rep(3, 4))`

4. Use the `rep()` function to define simply the following vectors in R.

   (a) `6, 6, 6, 6, 6, 6`
   (b) `5, 8, 5, 8, 5, 8, 5, 8`
   (c) `5, 5, 5, 5, 8, 8, 8, 8`

10.2 Summaries and Subscripting

1. If `x = c(5, 9, 2, 3, 4, 6, 7, 0, 8, 12, 2, 9)` decide what each of the following is and use R to check your answers:

   (a) `x[2]`
   (b) `x[2 : 4]`
   (c) `x[c(2, 3, 6)]`
   (d) `x[c(1 : 5, 10 : 12)]`
   (e) `x[−(10 : 12)]`

2. The data `y = c(33, 44, 29, 16, 25, 45, 33, 19, 54, 22, 21, 49, 11, 24, 56)` contains sales of milk in litres for 5 days in three different shops (the first 3 values are for shops 1,2, and 3 on Monday, etc.) Produce a statistical summary of the sales for each day of the week and also for each shop.

10.3 Matrices

1. Create in R the matrices

$$x = \begin{bmatrix} 3 & 2 \\ -1 & 1 \end{bmatrix}$$

and

$$y = \begin{bmatrix} 1 & 4 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Calculate the following expressions and check your answers in R:

(a) `2 * x`

(b) `x * x`

(c) `x %*% x`

(d) `x %*% y`

(e) `t(y)`

(f) `solve(x)`

2. With `x` and `y` as above, calculate the effect of the following subscript operations and check your answers in R.

(a) `x[1,]`

(b) `x[2,]`

(c) `x[,2]`

(d) `y[1,2]`

(e) `y[,2:3]`

10.4 The apply() function

1. If

$$y = \begin{bmatrix} 1 & 4 & 0 \\ 0 & 1 & -1 \end{bmatrix},$$

what is the result of `apply(y[,2:3],1,mean)`? Check your answer in R.

10.5 Statistical computation and simulation

1. Suppose $\texttt{X} \sim \texttt{N}(2, 0.25)$. Denote by `f` and `F` the density and distribution functions of `X`, respectively. Use R to calculate:

(a) `f(0.5)`

(b) `F(2.5)`

(c) $\texttt{F}^{-1}(0.95)$ (recall that $\texttt{F}^{-1}$ is the quantile function).

(d) $\texttt{Pr}(1 \leq \texttt{X} \leq 3)$

2. Use the function `rpois()` to simulate 100 values from a Poisson distribution with a parameter of your own choice. Produce a statistical summary of the result and check that the mean and variance are in reasonable agreement with the true population values.

3. Repeat the previous question replacing `rpois()` with `rexp()`.

10.6 Graphics

1. Use

`x = rnorm(100)`

or something similar to generate some data. Produce a figure showing a histogram and boxplot of the data. Modify the axis names and title of the plot in an appropriate way.

2. Type de following

   $\texttt{x} = (-10) : 10$

   $\texttt{n} = \texttt{length}(\texttt{x})$

   $\texttt{y} = \texttt{rnorm}(\texttt{n}, \texttt{x}, 4)$

   $\texttt{plot}(\texttt{x}, \texttt{y})$

   $\texttt{abline}(0, 1)$

   Try to understand the effect of each command and the graph that is produced.

3. Type de following

   $\texttt{data}(\texttt{nhtemp})$

   $\texttt{plot}(\texttt{nhtemp})$

   This produces a time series plot of measurements of annual mean temperatures in New Hampshire, USA.

4. The previous example illustrated that $\texttt{plot}()$ acts differently on objects of different types. More generally, we may have the data of yearly observations in a vector, but need to build the time series plot for ourselves. We can mimic this situation by typing

   $\texttt{temp} = \texttt{as.vector}(\texttt{nhtemp})$

   which creates a vector $\texttt{temp}$ that contains only the annual temperatures. We can produce something similar to the time series plot by typing

   $\texttt{plot}(1912 : 1971, \texttt{temp})$

   but this plots points rather than lines. To join the data via lines we would use

   $\texttt{plot}(1912 : 1971, \texttt{temp}, \texttt{type} =' \texttt{l}')$

   To get points and lines, use $\texttt{type} =' \texttt{b}'$ instead.

10.7 Writing functions

1. Write a fucntion that takes as its argument two vectors, $\texttt{x}$ and $\texttt{y}$, produces a scatterplot, and calculates the correlation coefficient (using $\texttt{cor}(\texttt{x}, \texttt{y})$).

2. Write a function that takes a vector $(\texttt{x}_1, \cdots, \texttt{x}_\texttt{n})$ and calculates both $\sum \texttt{x}_\texttt{i}$ and $\sum \texttt{x}_\texttt{i}^2$. (Remember the use of the function $\texttt{sum}()$).

11 References

1. RStudio Download, https://www.rstudio.com/products/rstudio/download/

2. The Comprehensive R Archive Network, https://cran.rstudio.com/

3. W.N. Venables, D.M. Smith and the R Core Team, "An Introduction to R", Notes on R: A Programming Environment for Data Analysis and Graphics Version 3.3.1 (2016-06-21)

4. National Center for Ecological Analysis and Synthesis, "R: A self-learn tutorial", Lecture Notes

5. The R Project for Statistical Computing, https://www.r-project.org/

6. Producting Simple Graphs with R, https://www.harding.edu/fmccown/r/

7. A Tutorial Introduction to R, https://kingaa.github.io/R_Tutorial/