This chapter deals with preparing R scripts to be run in a batch mode. We shall learn to control the output to the screen and input from the keyboard, to build conditional statements and loops as well as to program user-defined functions.

### 2.3 Coercion

In programming, one often faces the necessity to convert between different object types. In R exists a series of functions performing this task or testing the object mode.

- `as.numeric(x)`
- `as.character(x)`
- `as.expression(x)`

Try to coerce the object x to a given mode

- `as.matrix(x)`
- `as.data.frame(x)`
- `data.matrix(x)`

Attempt to convert an object x to matrix or data frame, respectively. A more user-friendly way of converting data frames to matrices is provided by the function `data.matrix` that converts all the variables in a data frame x to numeric mode and then binds them together as the columns of a matrix.

- `is.numeric(x)`
- `is.character(x)`
- `is.logical(x)`
- `is.matrix(x)`
- `is.data.frame(x)`

Test the given mode of the object x

### 2.4 Input and output

- `print(x)`

  Prints the contents of an object x.

- `cat(…, sep="")`

  Function `cat` displays contents of one or more R objects in somewhat less sophisticated way than `print`, but enables much larger control over the format of the output. Note that this function does not append a new line character explicitly and this must be added to the output string as "\n":

---

*Programming in R*
The result is 5.8 N/m.

This function provides also a possibility of writing to disc – see the R documentation.

**readline (prompt)**

Displays the prompt and then reads input from the keyboard:

```r
> x<-readline("Enter x:\n")
Enter x:
5.8
x
[1] "5.8"
```

From this example follows that the keyboard input is always in a form of a character vector of length 1. This needs be, if desired, coerced to a numeric value using the function `as.numeric`:

```r
> x<-as.numeric(x)
> x
[1] 5.8
```

**data.entry (x)**

Calls a built-in editor that can be used for entering/editing data in a spreadsheet-like form. It is invoked with a single parameter `x` specifying the name of R object to be edited.

### 2.5 Conditional execution

Conditional execution of R code can be achieved using:

```
if (condition) expression1 else expression2
```

In the case condition is fulfilled, expression1 is executed, else expression2 is run. More complicated commands may be grouped together in braces ("{}"):

```r
> if (x>2 & y<1){
>   print(x)
>   print(y)
> }else{
>   cat("x<=2 or y>=1!\n")
> }
```

### 2.6 Loops

Due to the powerful indexing engine of R, the loops are needed considerably less than in any conventional programming language such as BASIC or PASCAL. The loops can be built using the statement:

```
for (variable in expression1) expression2
```

Expression2 is executed for values of the control variable changing as specified by expression1. Again, more complicated commands must be grouped in braces.
For instance, the following for loop:

```r
> for (f in seq(1,10,by=2)){
>   cat("Square root of",f)
>   cat(" is",sqrt(f), "\n")
> }
```

produces an output:

Square root of 1 is 1
Square root of 3 is 1.732051
Square root of 5 is 2.236068
Square root of 7 is 2.645751
Square root of 9 is 3

Other, arguably less useful, commands for building loops are:

**while (condition) expression**

Expression will be executed until condition will not become invalid.

**repeat (condition) expression**

**NB:** Try to avoid loop commands if possible. Their execution in R tends to be rather tedious and there are most of the time other alternatives.

---

**Exercise 2.4**

- Using a loop and function `par(mfrow=)` write a short program that will plot six binary plots of SiO\(_2\) vs. other major-element oxides) of your choice. Use the data file `sazava.data`. Do not forget to label the axes as appropriate.

```r
> WR<-read.table("sazava.data",sep="\t")
> WR<-as.matrix(WR[,,-1])
> windows()
> par(mfrow=c(2,3))
> ee<-c("Al2O3","FeO","Fe2O3","MgO","CaO","K2O")
> for (f in ee){
>   plot(WR[,"SiO2"],WR[,f],xlab="SiO\(_2\)",ylab=f,pch=WR[,"Symbol"])
> }
```

Now we can prepare a more sophisticated version of this program, using annotations provided by the function `expression`. First of all we need to know that conversion of a character vector \(x\) to an expression can be done using the function `parse`:

```r
> parse(text=as.expression(x))
```

and then the modified part of the code will be:

```r
> lab<-c("Al\([2]\)*O\([3]\)","FeO","Fe\([2]\)*O\([3]\)","MgO","CaO","K\([2]\)*O")
> for (f in 1:length(ee)){
>   plot(WR[,"SiO2"],WR[,ee[f]],xlab="SiO\(_2\)",ylab=parse(text=as.expression(lab[f])),pch=WR[,"Symbol"])
> }
```

# Fig. 2.8
2.7 User-defined functions

The user-defined functions provide a stylish way of extending the set of the available commands. In fact, much of the R is itself written in R! The function definition looks like:

```
function.name <- function (argument1, argument2, …) expression
```

It is arbitrary that the last statement in the user-defined function body is `return(x)`, with the argument being a name of variable to be returned by the function\(^1\). Otherwise the last assigned one is assumed.

The concept of user-defined functions will be explained on a worked example, a user-defined function calculating a standard deviation following the formula:

\[
S = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}
\]  

(2.1)

```
> stdev <- function(x) {
>   z <- sqrt(sum((x-mean(x))^2)/length(x))
>   return(z)
> }
```

\(^1\) if more variables are to be returned, this can be done using an object list
Using the file `sazava.data`, we can calculate standard deviation for silica:

```r
> WR <- read.table("sazava.data", sep="\t")
> WR <- as.matrix(WR[, -1])
> stdev(WR[, "SiO2"])
[1] 7.462887
```

**Exercise 2.5** In some Anglo–Saxon countries, the temperature is still often expressed in degrees Fahrenheit. Their relation to the Centigrade is illustrated in Fig. 2.9.

- Prepare a function converting degrees of Fahrenheit to Centigrade
- Debug the function using the values provided in Fig. 4.2 — how much is in Celsius –40, 98.6 and 212 °F?
- Design a conversion table Fahrenheit → Celsius for 0, 10, …220 °F and plot the data into a binary diagram

```r
Fahr2Cels <- function(x) {
  a <- 100 / (212 - 32)
  b <- 100 - a * 212
  z <- a * x + b
  return(z)
}

Fahr2Cels(c(-40, 98.6, 212))
[1] -40  37 100

x <- seq(0, 220, by = 10)
y <- Fahr2Cels(x)
names(y) <- x
y
```

![Fig. 2.9. The relationship between Fahrenheit and Celsius scales](image)

```r
> plot(x, y, xlab = "Fahrenheit", ylab = "Celsius", type = "o")  # Fig. 2.10
```

![Fig. 2.10. Conversion Fahrenheit → Celsius (Exercise 2.5)](image)
2.7.1 Arguments to functions

There are two possibilities of providing arguments to a R function. First, you can pass the arguments in the order corresponding to that in the function’s definition. The second is to supply the arguments in the form “argument.name = value” in an arbitrary order.

When writing a user-defined function, one can provide default values like in the following example:

```r
my.plot<-function(x,y,pch="+",col="red"){
  ...
  function body...
  }
```

And such a function then can be called in a number of ways, for instance:

```r
my.plot(x,y)  # plots data as red crosses
my.plot(x,y,"o") # plots data as red circles
my.plot(x,y,col="blue") # plots data as blue crosses
```

But it is also obvious that:

```r
my.plot(x,y,"blue")
```

will not work correctly. Note that in R, most of the arguments have been given defaults appropriate to most of the applications. Thus ordinary user does not need to be aware of their existence at all. Command `args` lists arguments of a given function:

```
args (name)
```

where `name` refers to an existing function, e.g.:

```r
args("my.plot")
```

```r
function (x, y, pch = "+", col = "red")
```

2.7.2 Assignments in functions

Important is that the variables used within a user-defined function (in the worked example of function calculating standard deviations these were `x` and `z`) are local. This means that any assignments done within the function are temporary, being lost after exiting from the function. Therefore, such assignments do not affect the value of the variable with the same name in the calling program.

In (rare) cases when it is desirable to alter the value of a variable in the calling environment, this can be done by “super assignment” operator “<<-”:

```r
x <<- "Hello"
```