A Brief Introduction to R

Babak Shahbaba

Department of Statistics, University of California, Irvine, USA

Chapter 1
Introduction to R

1.1 Installing R

To install R, follow these steps:

1. Go to http://www.r-project.org/.
2. Click on the `download R` link.
3. Then select a location closest to you.
4. Click on your operating system (Linux, MacOS X, Windows) and follow directions

If you are a **Mac** user, download the latest .dmg file and follow the instructions. Once installed, R will appear as an icon in the Applications Folder. After you install R, you should go back to the same webpage (where you obtained the latest .dmg file), click on "tools", which is located under "Subdirectories", and install the universal build of **Tcl/Tk** for **X11**. The file name contains "tcltk", three numbers representing the current version, and "x11.dmg". (Currently, the file name is "tcltk-8.5.5-x11.dmg".) This file includes additional tools necessary for building R for Mac OS X.

If you are running Windows, click on `base` and then on the link that downloads R for Windows. (In the link, the current version number appears after "R".) When the dialog box opens, click `Run` and a "Setup Wizard" should appear. Keep clicking `Next` until the Wizard is finish. Now, you should see an icon on your desktop, with a large capital `R`.

1.2 Starting with R

In the R Console, you can type commands directly at the prompt, >, and execute them by pressing enter. The R Console provides an interactive environment where the results are immediately shown similar to a calculator. In fact, R can be used as a calculator. The basic arithmetic operators are + for addition, − for subtraction, ⋆

for multiplication and / for division. The ^ operator is used to raise a number (or a variable) to a power. Try executing the following commands.

```
> 65 + 32
[1] 97
> 3 * 1.7 - 2
[1] 3.1
> 4 * 3 + 6/0.2
[1] 42
> 5^2
[1] 25
```

There are also built in functions for finding the square root `sqrt()`, the exponential `exp()`, and the natural logarithm `log()`.

```
> sqrt(430)
[1] 20.73644
> exp(-1.3)
[1] 0.2725318
> log(25)
[1] 3.218876
```

Here, the numbers in the parentheses serve as input (parameters or arguments) to the functions. Most functions have multiple parameters and options. For example, to take the base-10 logarithm of 25, we include the option `base=10`

```
> log(25, base = 10)
[1] 1.39794
```

In general, you can always learn more about a function and its options (arguments) by writing its name after a question mark (e.g., `?log`).

We can combine two or more functions such that the output from one function becomes the input for another function. For example, the following code combines the above `log()` function with the `round()` function, which is used to specify the number of decimal places (here, two digits).

```
> round(log(25, base = 10), digits = 2)
[1] 1.4
```

In the above code, the output of `log(25, base=10)` becomes the first argument of the function `round()`. The second argument, `digits=2`, specifies the number of decimal places.

1.3 Creating objects in R

Instead of directly entering commands such as `2+3`, we can create *objects* to hold values and then perform operations on these objects. For example, the following set of commands creates two objects `x` and `y`, adds the values stored in these objects, and assigns the result to the third object `z`

```
> x <- 2
> y <- 3
> z <- x + y
```

In general, we use left arrow <− (i.e., type < and then −) to assign values to an object. Almost always, we can use the equal sign "=" instead of <− for assignment. For example, we could use `y = 3`) to assign 3 to *y*, and use `z = x+y` to set *z* equal to the sum of *x* and *y*. However, the two options, "=" and <−, have some small differences, which are not discussed here.

Simply typing the name of an object displays its contents. We could also use the function `print()`.

```
> x

[1] 2

> print(y)

[1] 3

> print(z)

[1] 5
```

Object names are case sensitive. For example, `x` and `X` are two different objects. A name cannot start with a digit or an underscore _ and cannot be among the list of reserved words such as `if, function, NULL`. We can use a period `.` in a name to separate words (e.g., `my.object`).

The objects are created and stored by their names. We can obtain the list of objects that are currently stored within R by using the function `objects()`.

```
> objects()

[1] "x" "y" "z"
```

The collection of these objects is called the *workspace*. (Note that although we are not specifying the values of arguments, we still need to type parentheses when using functions.) When closing an R session, you have the opportunity to save the objects permanently for future use. This way, the workspace is saved in a file called `.RData`, which will be restored next time you open R. If you want to save only few objects, as opposed to the entire workspace, you can use the function `save()`. For example, suppose we only want to save the objects `x` and `y`.

```
> save(x, y, file = "myObjects.RData")
```

The above command, saves `x` and `y` in a file called `myObjects.RData` in the *current working directory*. You can see where the current working directory is by entering the command `getwd()`. If you want to save your objects in another directory, either enter the full path when specifying the file name in the `save()` function, or use the menu bar to change the directory. (For Mac, this option is located under "Misc". For Windows, it is located under "File".)

After you save the `x` and `y` in `myObjects.RData`, you can load these objects for future use with the `load()` function.

```
> load("myObjects.RData")
```

As before, give the full address for the file if it is not located in the current working directory. Alternatively, you can change the working directory from the menu bar.

1.4 Vectors

Using objects allows for more flexibility. For example, we can store more than one value in an object and apply a function or an operation to its contents. The following commands create a *vector* object `x` that contains numbers 1 through 5, and then apply two different functions to it.

```
> x <- c(1, 2, 3, 4, 5)
> x

[1] 1 2 3 4 5

> 2 * x + 1

[1]  3  5  7  9 11

> exp(x)

[1]   2.718282   7.389056  20.085537  54.598150
[5] 148.413159
```

The `c()` function is used to combine its arguments (here, integers from 1 to 5) into a vector. Since $1, 2, \ldots, 5$ is a sequence of consecutive numbers, we could simply use the colon ":" operator to create the vector.

```
> x <- 1:5
> x

[1] 1 2 3 4 5
```

To create sequences and store them in vector objects, we can also use the `seq()` function for additional flexibility. The following commands creates a vector object `y` containing a sequence increasing by 2 from -3 to 14.

```
> y <- seq(from = -3, to = 14, by = 2)
> y

[1] -3 -1  1  3  5  7  9 11 13
```

If the elements of a vector are all the same, we can use the `rep()` function.

```
> z <- rep(5, times = 10)
> z

 [1] 5 5 5 5 5 5 5 5 5 5
```

The following function creates a vector of size 10 where all its elements are unknown. In R, missing values are represented by `NA` (Not Available).

```
> z <- rep(NA, times = 10)
> z

 [1] NA NA NA NA NA NA NA NA NA NA
```

This way, we can create a vector object of a given length and specify its elements later.

To find the length of a vector (i.e., number of elements), we use the `length()` command.

```
> length(x)

[1] 5

> length(y)

[1] 9
```

Functions `sum(), mean(), min()` and `max()` return the sum, average, minimum, and maximum for a vector.

```
> x

[1] 1 2 3 4 5

> sum(x)

[1] 15

> mean(x)

[1] 3

> min(x)

[1] 1

> max(x)
```

```
[1] 5
```

The elements of a vector can be accessed by providing their index using square brackets `[ ]`. For example, try retrieving the first element of `x` and the 4^{th} element of `y`.

```
> x[1]

[1] 1

> y[4]

[1] 3
```

The colon `:` operator can be used to obtain a sequence of elements. For instance, elements 3 through 6 of `y` can be accessed with

```
> y[3:6]

[1] 1 3 5 7
```

To select all but the 4^{rd} element of a vector, we use negative indexing.

```
> y[-4]

[1] -3 -1  1  5  7  9 11 13
```

The above objects are all *numerical* vectors. A vector can also hold character strings delimited by single or double quotations marks. For example, suppose we have a sample of 5 patients. We can create a *character* vector storing their gender as

```
> gender <- c("male", "female", "female", "male",
+     "female")
```

(The plus sign means that the second line is the continuation of the command in the first line. You should not type it when trying the above command in R.) Retrieving the elements of the vector is as before.

```
> gender[3]

[1] "female"
```

A vector could also be *logical*, where the elements are either `TRUE` or `FALSE` (`NA` if the element is missing). Note that these values must be in capital letters and can be abbreviated by `T` and `F` (not recommended). For example, create a vector storing the health status of the five patients.

```
> is.healthy <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
> is.healthy

[1]  TRUE  TRUE FALSE  TRUE FALSE
```

When used in ordinary arithmetic, logical vectors are coerced to integer vectors, 0 for FALSE elements and 1 for TRUE elements. For example, applying the sum() function to the above logical vector turns the TRUE and FALSE values to ones and zeros, and returns their sum, which in this case is equivalent to the number of healthy subjects.

```
> sum(is.healthy)

[1] 3
```

Use the as.integer() function to see the equivalent integer vector for is.healthy.

```
> as.integer(is.healthy)

[1] 1 1 0 1 0
```

Logical vectors are usually derived from other vectors using logical operators. For example, with the gender vector, we can create a logical vector showing which subjects are male.

```
> gender

[1] "male"   "female" "female" "male"   "female"

> is.male <- (gender == "male")
> is.male

[1]  TRUE FALSE FALSE  TRUE FALSE
```

Here, == (i.e. two equal signs) is a relational operator that returns TRUE if the two sides are equal and returns FALSE otherwise. As the second example, we create a numerical vector for the age of the five subjects and then check to see which person is 60 years old.

```
> age <- c(60, 43, 72, 35, 47)
> is.60 <- (age == 60)
> is.60

[1]  TRUE FALSE FALSE FALSE FALSE
```

The != operator, on the other hand, returns a TRUE value when the two sides are not equal:

```
> is.female <- (gender != "male")
> is.female

[1] FALSE  TRUE  TRUE FALSE  TRUE

> not.60 <- (age != 60)
> not.60

[1] FALSE  TRUE  TRUE  TRUE  TRUE
```

The other relational operators commonly applied to numerical vectors are "less than", <, "less than or equal to", <=, "greater than", >, "greater than or equal to", >=

```
> age < 43

[1] FALSE FALSE FALSE  TRUE FALSE

> age <= 43

[1] FALSE  TRUE FALSE  TRUE FALSE

> age > 43

[1]  TRUE FALSE  TRUE FALSE  TRUE

> age >= 43

[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

We can also use *Boolean* operators to create new logical vectors based on existing ones. The logical NOT, `!`, negates the elements of a logical vector (i.e., changes `TRUE` to `FALSE` and vice versa). For example, create a `is.female` vector from the `is.male` vector:

```
> is.female <- !is.male
> is.female

[1] FALSE  TRUE  TRUE FALSE  TRUE
```

The logical AND, `&`, compares the elements of two logical vectors, and returns `TRUE` only when the corresponding elements are both `TRUE`:

```
> is.male

[1]  TRUE FALSE FALSE  TRUE FALSE

> is.healthy

[1]  TRUE  TRUE FALSE  TRUE FALSE

> is.male & is.healthy

[1]  TRUE FALSE FALSE  TRUE FALSE
```

The logical OR, `|`, also compares the elements of two logical vectors, and returns `TRUE` when at least one of the corresponding elements is `TRUE`:

```
> is.male | is.healthy

[1]  TRUE  TRUE FALSE  TRUE FALSE
```

We can use combinations of two or more logical operators. The following commands check to see which subjects are male and less than 45 years old.

```
> is.young.male <- is.male & (age < 45)
> is.young.male

[1] FALSE FALSE FALSE  TRUE FALSE
```

Using the `which()` function, we can obtain the indices of `TRUE` elements for a given logical function.

```
> ind.male <- which(is.male)
> ind.male

[1] 1 4

> ind.young <- which(age < 45)
> ind.young

[1] 2 4
```

We can then use these indices to obtain their corresponding elements from a vector.

```
> age[ind.male]

[1] 60 35

> is.male[ind.young]

[1] FALSE  TRUE
```

We can combine the two steps.

```
> age[is.male]

[1] 60 35

> age[gender == "male"]

[1] 60 35

> is.male[age < 45]

[1] FALSE  TRUE

> gender[age < 45]

[1] "female" "male"
```

1.5 Matrices

In the above examples, we used one vector for each characteristic (e.g., age, health status, gender) of our five subjects. It is easier, of course, to store the subject information in a table format, where each row corresponds to an individual and each column to a characteristic. If all these measurements are from the same type (e.g., numerical, character, logical), a *matrix* can be used. For example, besides age, assume that for our five subjects we have also measured BMI (body mass index) and blood pressure:

```
> BMI = c(28, 32, 21, 27, 35)
> bp = c(124, 145, 127, 133, 140)
```

Now create a matrix with the `cbind()` function for column-wise binding of `age`, `BMI`, and `bp`.

```
> data.1 = cbind(age, BMI, bp)
> data.1
```

```
     age BMI  bp
[1,]  60  28 124
[2,]  43  32 145
[3,]  72  21 127
[4,]  35  27 133
[5,]  47  35 140
```

If we had wanted a matrix where each row represented a characteristic and each column a subject, we would have used the `rbind()` function for row-wise binding.

```
> data.2 = rbind(age, BMI, bp)
> data.2
```

```
    [,1] [,2] [,3] [,4] [,5]
age   60   43   72   35   47
BMI   28   32   21   27   35
bp   124  145  127  133  140
```

We could obtain `data.2` by transposing (e.g., interchanging the rows and columns) `data.1` using the `t` function:

```
> t(data.1)
```

```
    [,1] [,2] [,3] [,4] [,5]
age   60   43   72   35   47
BMI   28   32   21   27   35
bp   124  145  127  133  140
```

In general, matrices are two dimensional objects comprised of values of the same type. The object `data.1` is a 5×3 matrix. The function `dim` returns the size (i.e., the number of rows and columns) of a matrix:

```
> dim(data.1)

[1] 5 3
```

When creating the matrix `data.1`, R automatically uses the vector names as the column names. They can be changed or accessed with the function `colnames()`.

```
> colnames(data.1)

[1] "age" "BMI" "bp"
```

Likewise, we can obtain or provide the row names using the function `rownames()`.

```
> rownames(data.1) <- c("subject1", "subject2",
+     "subject3", "subject4", "subject5")
> data.1

         age BMI  bp
subject1  60  28 124
subject2  43  32 145
subject3  72  21 127
subject4  35  27 133
subject5  47  35 140
```

To access the elements of a matrix, we still use square brackets `[ ]`, but this time, we have to provide both the row index and the column index. For instance, the age of the third subject is

```
> data.1[3, 1]

[1] 72
```

If only a row number is provided, R returns all elements of that row (e.g., all the measurements for one subject):

```
> data.1[2, ]

age BMI  bp
 43  32 145
```

Likewise, if only a column is specified, R returns all elements of that column (e.g., all the measurements for one characterstic):

```
> data.1[, 2]

subject1 subject2 subject3 subject4 subject5
      28       32       21       27       35
```

A matrix can also be created by re-arranging the elements of a vector with the `matrix` function:

```
> matrix(data = 1:12, nrow = 3, ncol = 4)
```

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

Here, the argument `data` specifies the numbers (vector), whose re-arrangement creates the matrix. The arguments `nrow` and `ncol` are the number of rows and columns, respectively. By default, the matrix is filled by columns. To fill the matrix by rows, we must use the argument `byrow=TRUE`:

```
> matrix(data = 1:12, nrow = 3, ncol = 4, byrow = TRUE)
```

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

The length of `data` is usually equal to $nrow \times ncol$. If there are too few elements in `data` to fill the matrix, then the elements are recycled. In the following example, all elements of the matrix are set to zero.

```
> mat <- matrix(data = 0, nrow = 3, ncol = 3)
> mat
```

```
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

We can obtain or set the diagonal elements of a matrix using the `diag()` function.

```
> diag(mat) <- 1
> mat
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Similar to vectors, we can create a matrix with missing values (NA) and specify the elements later.

```
> mat <- matrix(data = NA, nrow = 2, ncol = 3)
> mat
```

```
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

```
> mat[1, 3] <- 5
> mat
```

```
     [,1] [,2] [,3]
[1,]   NA   NA    5
[2,]   NA   NA   NA
```

1.6 Data frames

In general, we use matrices when all measurements are of the same type (e.g., numerical, character, logical). Otherwise, the type of measurements could change when we mix different types. In the following example, we show this using the `mode()` function that returns the type for a given object.

```
> mat <- cbind(age, gender, is.healthy)
> mode(mat)

[1] "character"
```

In the above example, the matrix type is character, although `age` is numeric and `is.healthy` is logical. Note that their values are now printed in quotation marks, which is used for character strings.

To avoid this issue, we can store data with information of different types in a table format similar to the format of spreadsheets (e.g., Excel). The resulting object (which includes multiple objects of possibly different types) is called a *data frame* object. For this, we use the function `data.frame()`.

```
> data.df = data.frame(age, gender, is.healthy,
+     BMI, bp)
> data.df

  age gender is.healthy BMI  bp
1  60   male       TRUE  28 124
2  43 female       TRUE  32 145
3  72 female      FALSE  21 127
4  35   male       TRUE  27 133
5  47 female      FALSE  35 140
```

To create a data frame this way, all vectors must have the same length. You may notice that while `gender` is a character vector, its elements are not printed with quotation marks as before. The reason is character vectors included in data frames are coerced to a different type called *factors*. A factor is a vector object that is used to provide a compact way to represent categorical data. Each *level* of a factor vector represents a unique category (e.g., female or male). We can directly create factors using the `factor()` function.

```
> factor(gender)

[1] male   female female male   female
Levels: female male
```

To access elements of a data frame, we use the square brackets `[ , ]` with the appropriate row and column indices. For example, the BMI of the 3^{rd} subject is

```
> data.df[3, 4]

[1] 21
```

As before, we can access an entire row (e.g., all the measurements for one subject) by only specifying the row index, and an entire column (e.g., all the measurements for one variable) by only specifying the column index. We can also access an entire column by providing the column name.

```
> data.df[, "age"]

[1] 60 43 72 35 47
```

The $ operator also retrieves an entire column from the data frame.

```
> data.df$age

[1] 60 43 72 35 47
```

This column can then be treated as a vector and its elements accessed with the square brackets as before. For instance, try obtaining the `BMI` for the 3^{th} subject and the `gender` of the 2^{nd} subject.

```
> data.df$BMI[4]

[1] 27

> data.df$gender[2]

[1] female
Levels: female male
```

1.6.1 Creating data frames using a spreadsheet-like environment

We can create a data frame by invoking a spreadsheet-like environment in R. For this, we start by creating an empty data frame object.

```
> new.df <- data.frame()
```

Then, we use the function `fix()` to edit the newly created data frame `new.df`.

```
> fix(new.df)
```

This way, R opens a window for data editing similar to Fig. 1.1. You can use the four icons at the top of the editor to add or delete columns or rows. (Hover your mouse pointer over each icon to see its function.) For `new.df`, we have created four columns and three rows. The column names and the content of the data frame can be edited the same way we edit spreadsheets.

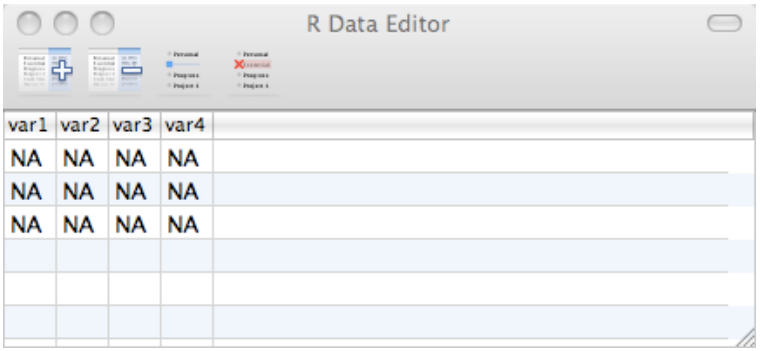

Fig. 1.1 R window for data editing invoked by applying the function `fix()` to an empty data frame object.

1.6.2 Importing data from text files

Data are usually available in a tabular format as a delimited text file. We can import the contents of such files into R using the function `read.table()`. For instance, let us try importing the `BodyTemperature` data set. (This data set is available online from our website.)

```
> BodyTemperature <- read.table(file = "BodyTemperature.txt",
+     header = TRUE, sep = " ")
```

Here, we are using the `read.table()` function with three arguments. The first argument, `file="BodyTemperature.txt"`, specifies the name and location of the data file. If the file is not in the current working directory, you need to give the full path to the file or change the working directory. The `header=TRUE` option tells R that the variable names are contained in the first line of the data. Set this option to `FALSE` when this is not the case. The `sep=" "` option tells R that the columns in the data set are separated by white spaces. If the columns were separated by commas, we would have used `sep=","`.

The `BodyTemperature` object is a data frame, holding the contents of the "BodyTemperature.txt" file. Type `BodyTemperature` to view the entire data set. If the data set is large, it is better to use the `head()` function, which shows only the first part (few rows) of the data set.

```
> head(BodyTemperature)

  Gender Age HeartRate Temperature
1      M  33        69        97.0
2      M  32        72        98.8
3      M  42        68        96.2
4      F  33        75        97.8
5      F  26        68        98.8
6      M  37        79       101.3
```

In the `BodyTemperature` data frame, the rows correspond to subjects and the columns to variables. To view the names of the columns, try

```
> names(BodyTemperature)
```

```
[1] "Gender"      "Age"         "HeartRate"
[4] "Temperature"
```

Accessing observations in the `BodyTemperature` data frame is the same as before. We can use square brackets `[ , ]` with the row and column indices or the $ operator with the variable name.

```
> BodyTemperature[1:3, 2:4]
```

```
  Age HeartRate Temperature
1  33        69        97.0
2  32        72        98.8
3  42        68        96.2
```

```
> BodyTemperature$Age[1:3]
```

```
[1] 33 32 42
```

1.7 Lists

The data frames we created above (either directly or by importing a text file) includes vectors of different types, but all the vectors have the same length. To combine objects of different types and possibly with different length into one object, we use *lists* instead. For example, suppose we want to store the above body temperature data along with the name of investigators and students who have been involved in the study. We can create a list as follows:

```
> our.study <- list(data = BodyTemperature,
+     investigators = c("Smith", "Jackson",
+         "Clark"), students = c("Steve", "Mary"))
> length(our.study)
```

```
[1] 3
```

We created a list with three *components*: `data`, `investigators`, and `students`. Each component has a name, which can be used to access that component.

```
> our.study$investigator
```

```
[1] "Smith"   "Jackson" "Clark"
```

The components are ordered, so we can access them using a double square brackets: "[[ ]]".

```
> our.study[[2]]

[1] "Smith"   "Jackson" "Clark"
```

If the component is a matrix or a vector, we can access its individual elements as before.

```
> our.study[[2]][3]

[1] "Clark"

> our.study[[1]][2:4, ]

  Gender Age HeartRate Temperature
2      M  32        72        98.8
3      M  42        68        96.2
4      F  33        75        97.8
```

1.8 Loading add-on packages

A package includes a set of functions that are commonly use for specific area of statistical analysis. R users constantly create new packages and make them publicly available on CRAN (Comprehensive R Archive Network). To use a package, you first need to download the packages from CRAN and install it in your local R library. For this, we can use the `install.packages()` function. For example, suppose we want to perform Biodemographic analysis. The "Biodem" package, which is created by Boattini et. al., provides a number of functions for this purpose. The following command downloads the `Biodem` package.

```
> install.packages("Biodem", dependencies = TRUE)
```

The first argument specifies the name of the package, and by setting the option `dependencies` to "TRUE", we install all other packages on which "Biodem" depends. The reference manual for this package is available at http://cran.r-project.org/web/packages/Biodem/Biodem.pdf.

After we install a package, we need to load it in R in order to use it. For this, we use the `library()` command.

```
> library(Biodem)
```

Now we can use all the functions available in this package. We can also use all the data sets included in the package. For example, the `Biodem` package includes a data set called `valley` where every row corresponds to a different marriage record. To use this data set, we enter the following command

```
> data(valley)
```

The data set becomes available in the workspace as a data frame.

1.9 Conditional statements

One of the most widely used packages in R is the `MASS` package. This package is automatically installed when you install R. However, you still need to load it before you can use it. Enter the command `library(MASS)` to load this package. One of the data sets available in the `MASS` package is the `birthwt` data set, which includes the birthweight, `bwt`, of newborn babies. It also includes a binary variable, `low`, that indicates whether the baby had low birthweight. Low birthweight is defined as having birthweight lower than 2500 grams (2.5 kilograms). Suppose we did not have this variable and we wanted to create it. First, let us load the `birthwt` data set into R.

```
> data(birthwt)
> dim(birthwt)

[1] 189  10
```

The data set includes 189 cases and 10 variables.

We now create an empty vector, called `low`, of size 189.

```
> low <- rep(NA, 189)
```

Alternatively, we could use create an empty numerical vector without specifying its length using the `numeric()` function.

```
> low <- numeric()
```

Note that this way, we specify the type of the object. We would have used `character()`, or `data.frame()`, or `list()`, if we wanted to create an empty object of the type character, or data frame, or list.

Now we want to examine the birthweight of each baby, and *if* it is below 2500, we assign the value of "1" to the `low` variable, otherwise, we assign the value "0". The general format for an `if()` statement is

```
> if (condition) {
+     expression
+ }
```

If the `condition` is true, R runs the commands represented by `expression`. Otherwise, R skips the commands within the brackets { }.

Try an `if()` statement to set the `low` of the first observation.

```
> if (birthwt$bwt[1] < 2500) {
+     low[1] <- 1
+ }
```

Check the result:

```
> birthwt$bwt[1]

[1] 2523
```

```
> low[1]

[1] NA
```

Since the `condition` was not true (i.e, `bwt` is not below 2500), the `expression` was not executed. To assign the value "0", we can use an `else` statement along with the above `if` statement. The general format for `if-else()` statements is

```
> if (condition) {
+     expression1
+ } else {
+     expression2
+ }
```

If the `condition` is true, R runs the commands represented by `expression1`; otherwise, R runs the commands represented by `expression2`. For example, we can use the following code to decide whether the first baby in the `birthwt` data has low birthweight or not:

```
> if (birthwt$bwt[1] < 2500) {
+     low[1] <- 1
+ } else {
+     low[1] <- 0
+ }
> birthwt$bwt[1]

[1] 2523

> low[1]

[1] 0
```

Conditional statements can have multiple `else` statements to test multiple conditions.

```
> if (condition1) {
+     expression1
+ } else if (condition2) {
+     expression2
+ } else {
+     expression3
+ }
```

1.10 Loops

To apply the above conditional statements to all observations, we can use a `for()` loop, which has the general format

```
> for (i in 1:n) {
+     expression
+ }
```

Here, i is the loop counter that takes values from 1 through n. The `expression` within the loop represents the set of commands to be repeated n times. For example, the following R commands create the vector y based on the vector x one element at a time.

```
> x <- c(3, -2, 5, 6)
> y <- numeric()
> for (i in 1:4) {
+     y[i] <- x[i] + 2 * i
+ }
> y
```

```
[1]  5  2 11 14
```

For the example discussed in the previous section, we use the following loop:

```
> for (i in 1:189) {
+     if (birthwt$bwt[i] < 2500) {
+         low[i] <- 1
+     }
+     else {
+         low[i] <- 0
+     }
+ }
```

The counter starts from 1 (i.e., the first row) and it ends at 189 (i.e., the last row). At each iteration, evaluate the conditional expression `birthwt$bwt[i] < 2500`. If the expression is true, it set the value of `low` for that row to 1, otherwise, it sets it to 0. The variable `low` variable, which you created using the above loop and conditional statements, will be exactly the same as the existing variable `low` in the data frame `birthwt`.

1.11 Creating functions

So far, we have been using R to perform specific tasks by creating objects and applying functions to them. If we need to repeat the same task over an over again under different setttings, a more efficient approach would be to create a *function*, which can be called repeatedly. The function we create itself is an object and is similar to existing functions in R, such as `sum()`, `log()`, and `matrix()`, that we have been using. The general form of creating a function is as follows:

```
> fun.name <- function(arg1, arg2, ...) {
+     expression1
```

```
+     expression2
+     ...
+     return(list = c(out1 = output1, out2 = output2, ...))
+ }
```

For example, suppose we routinely need to find the min and max for a given numerical vector, and print the sum of its elements. Also, we need to round the elements of the vector. However, the number of decimal places could be different for different vectors. Instead of writing the codes to create the function in *R Console*, it is better to write it in a file using a text editor so we can modify it later. For this, click `File` → `New Document` from the menu bar. This will open a text editor. Now we can type the following commands in the text editor to create our function (Fig. 1.2).

```
> my.fun <- function(x, n.digits = 1) {
+     min.value <- min(x)
+     max.value <- max(x)
+     print(sum(x))
+     y <- round(x, digits = n.digits)
+     return(list(min.value = min.value,
+         max.value = max.value, rounded.vec = y))
+ }
```

The above function takes two inputs: a numerical vector *x*, and the number of decimal places `n.digits`. For the number of decimal places, we set the default to 1. If the user does not specify the number of decimal points, the function uses the default value.

The function then creates two objects, `min.value` and `max.value`, that store the min and max of `x` respectively. Next, the function prints the sum of all elements. Finally, the function create a new vector called `y`, which contains the rounded values of the original vector to the number of decimal place specified by `n.digits`.

Using `return()`, we specify the outputs of the function as a list. In this case, the list has three components. The first component is called "min.value" and it contains the value of the object `min.value`. The second component is called "max.value" and it contains the value of the object `max.value`. The last component is called "rounded.vec", and it contains the new vector, `y`, which was created by rounding the values of the original vector.

Note that in Fig. 1.2, we wrote some comments in the text editor to explain what the function does. The comments should be always followed by the symbol "#". R regards what we write after "#" as comments and does not execute them.

When we finish typing the commands required to create the function, we save the file by clicking `File` → `Save As`. When prompted, choose a name for your file. For example, we called our file "CreateMyFun.R". The file will have a ".R" extension.

So far, we have just created a file that contains the command necessary to create the function. The function has not been created yet. To create the function, we need

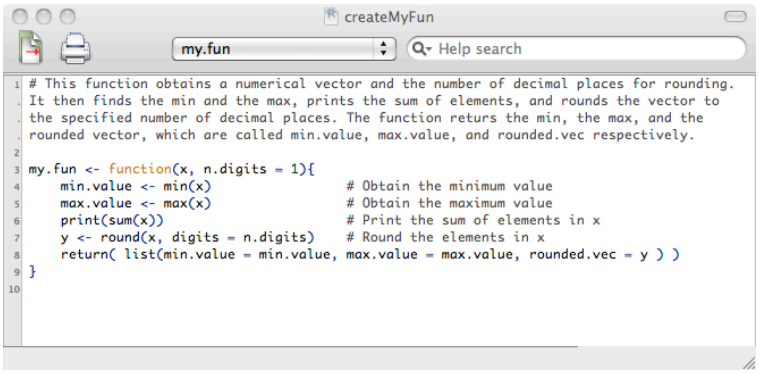

Fig. 1.2 Creating a function called `my.fun()` using the text editor in R.

to execute the commands. For this, we can use the `source()` function to read (evaluate) the codes from the "CreateMyFun.R" file.

```
> source("CreateMyFun.R")
```

Again, give the full address for the file if it is not located in the current working directory. We can now use our function the same way we have been using any other function. The following is an example.

```
> out <- my.fun(x = c(1.2, 2.4, 5.7), n.digits = 0)

[1] 9.3

> out

$min.value
[1] 1.2

$max.value
[1] 5.7

$rounded.vec
[1] 1 2 6
```

When we run the function, it prints the sum of all elements, which is 9.3, as we requested. The outputs will be assigned to a new object called "out". Since the output was a list, `out` will be a list, and we can print its contents by entering its name.