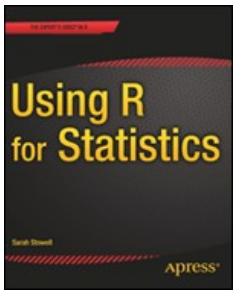


Using R for Statistics

by Sarah Stowell

Chapter 1: R Fundamentals

Overview

R is a statistical analysis and graphics environment that is comparable in scope to the SAS, SPSS, Stata, and S-Plus packages. The basic installation includes all of the most commonly used statistical techniques such as univariate analysis, categorical data analysis, hypothesis tests, generalized linear models, multivariate analysis, and time-series analysis. It also has excellent facilities for producing statistical graphics. Anything not included in the basic installation is usually covered by one of the thousands of add-on packages available.

Because R is command-driven, it can take a little longer to master than point-and-click style software. However, the reward for your effort is the greater flexibility of the software and access to the most newly developed methods.

To get you started, this chapter introduces the R system. You will:

- download and install R
- become familiar with the interface
- start giving commands
- learn about the different types of R files
- become familiar with all of the important technical terms that will be used throughout the book

If you are new to R, I recommend that you read the entire chapter, as it will give you a solid foundation on which to build.

Downloading and Installing R

The R software is freely available from the R website. Windows® and Mac® users should follow the instructions below to download the installation file:

1. Go to the R project website at www.r-project.org.
2. Follow the link to CRAN (on the left-hand side).
3. You will be taken to a list of sites that host the R installation files (mirror sites). Select a site close to your location.
4. Select your operating system. There are installation files available for the Windows, Mac, and Linux® operating systems.
5. If downloading R for Windows, you will be asked to select from the base or contrib distributions. Select the base distribution.
6. Follow the link to download the R installation file and save the file to a suitable location on your machine.

To install R for the Windows and Mac OS environments, open the installation file and follow the instructions given by the setup wizard. You will be given the option of customizing the installation, but if you are new to R, I recommend that you use the standard installation settings. If you are installing R on a networked computer, you may need to contact your system administrator to obtain permission before performing the installation.

For Linux users, the simplest way to install R is via the package manager. You can find R by searching for "r-base-core." Detailed installation instructions are available in the same location as the installation files.

If you have the required technical knowledge, then you can also compile the software from the source code. An in-depth guide can be found at www.stats.bris.ac.uk/R/doc/manuals/R-admin.pdf.

Getting Orientated

Once you have installed the software and opened it for the first time, you will see the R interface as shown in Figure 1-1.

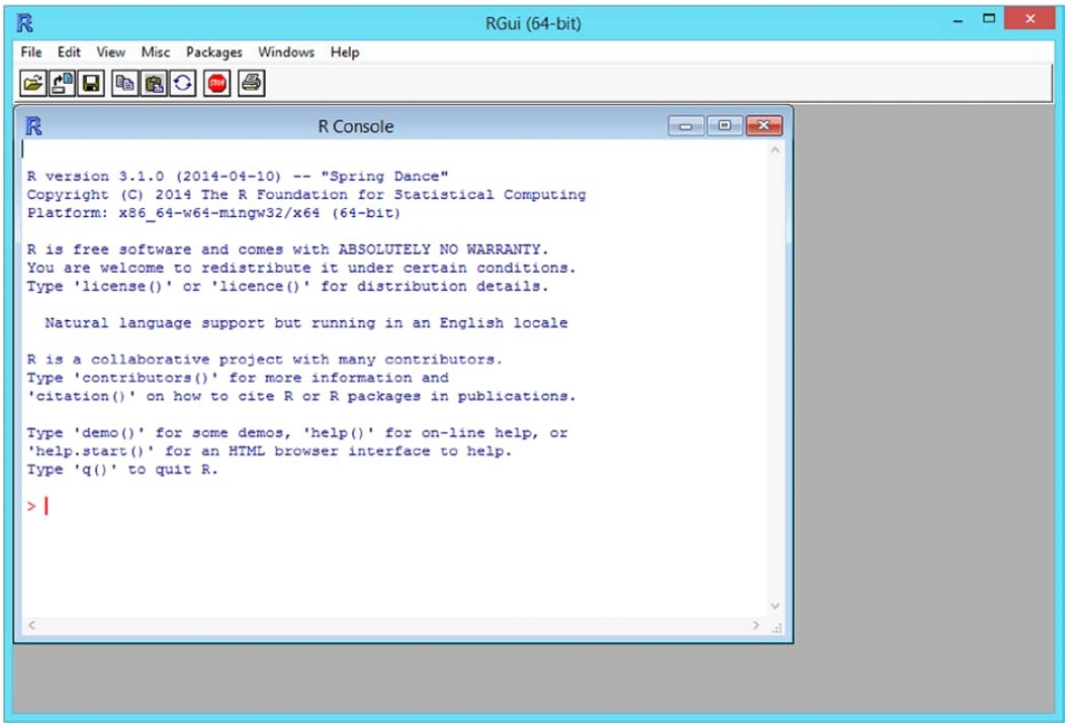


Figure 1-1: The R interface

There are several drop-down menus and buttons, but unlike in point-and-click style statistical packages, you will only use these for supporting activities such as opening and saving R files, setting preferences, and loading add-on packages. You will perform all of the main tasks (such as importing data, performing statistical analysis, and creating graphs) by giving R typed *commands*.

The R Console window is where you will type your commands. It is also where the output and any error messages are displayed. Later you will use other windows such as the data editor, script editor, and graphics device.

The R Console and Command Prompt

Now turn your attention to the R console window. Every time you start R, some text relating to copyright and other issues appears in the console window, as shown in Figure 1-1. If you find the text in the console difficult to read, you can adjust it by selecting GUI Preferences from the Edit menu. This opens a dialog box that allows you to change the size and font of the console text, as well as other options.

Below all of the text that appears in the console at startup you will see the *command prompt*, which is colored red and looks like this:

```
>
```

The command prompt tells you that R is ready to receive your command.

Try typing the following command at the prompt and pressing Enter:

```
> 8-2
```

R responds by giving the following output in the next line of the console:

```
[1] 6
>
```

The `[1]` tells you which component of the output you are looking at, which is not of much interest at this stage as the output has only one component. This is followed by the result of the calculation, which is 6. Notice that all output is shown in blue, to distinguish it from your commands.

The output is followed by another prompt > to tell you that it has finished processing your command and is ready for the next one. If you don't see a command prompt after entering a command, it may be because the command you have given is not complete. Try entering the following incomplete command at the command prompt:

```
> 8-
```

R responds with a plus sign:

```
+
```

If you see the plus sign, it means you need to type the remainder of the command and press Enter. Alternatively, you can press the Esc key to cancel the command and return to the command prompt.

Another time that you would not see the command prompt is when R is still working on the task. Usually this time is negligible, but there may be some waiting time for more complex tasks or those involving large datasets. If a command takes much longer than expected to complete, you can cancel it with the Esc key.

From here onward, the command prompt will be omitted when showing output.

Table 1-1 shows the symbols used to represent the basic arithmetic operations.

Table 1-1: Arithmetic Operators

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	^

If a command is composed of several arithmetic operators, they are evaluated in the usual order of precedence, that is, first the exponentiation (power) symbol, followed by division, then multiplication, and finally addition and subtraction. You can also add parentheses to control precedence if required. For example, the command:

```
> 3^2+6/3+2
```

gives the result:

```
[1] 13
```

while the command:

```
> (3^2+6)/(3+2)
```

gives the result:

```
[1] 3
```

If you want to repeat a command, you can use the up and down arrow keys on your keyboard to scroll through previously entered commands. You will be able to edit the command before pressing Enter. This means that you don't have to retype a whole command just to correct a minor mistake, which you will find useful as you begin to use longer and more complex commands.

Functions

In order to do anything more than basic arithmetic calculations, you will need to use *functions*. A function is a set of commands that have been given a name and together perform a specific task producing some kind of output. Usually a function also requires some kind of data as input.

R has many built-in functions for performing a variety of tasks from simple things like rounding numbers, to importing files and performing complex statistical analysis. You will make use of these throughout this book. You can also create your own functions, which is covered briefly in Chapter 12.

Whenever you use a function, you will type the function name followed by round brackets. Any input required by the function is placed between the brackets.

An example of a function that does not require any input is the `date` function, which gives the current date and time from your computer's clock.

```
> date()
```

```
[1] "Thu Apr 10 20:59:26 2014"
```

An example of a simple function that requires input is the `round` function, which rounds numbers. The input required is the number you want to round. A single piece of input is known as an *argument*.

```
> round(3.141593)
```

```
[1] 3
```

As you can see, the `round` function rounds a given number to the nearest whole number, but you can also use it to round a number to a different level of accuracy. The command below rounds the same number to two decimal places:

```
> round(3.141593, digits=2)
```

```
[1] 3.14
```

We were able to change the behavior of the `round` function by adding an additional argument giving the number of decimal places required. When you provide more than one argument to a function, they must be separated with commas. Each argument has a name. In this case, the argument giving the number of decimal places is called `digits`. Often you don't need to give the names of the arguments, because R is able to identify them by their values and the order in which they are arranged. So for the `round` function, the following command is also acceptable:

```
> round(3.141593, 2)
```

Some arguments are optional and some must be provided for the function to work. For the `round` function, the number to be rounded (in this example 3.141593) is a required argument and the function won't work without it. The `digits` argument is optional. If you don't supply it, R assumes a default value of zero.

For every function included with R, there is a help file that you can view by entering the command:

```
> help(functionname)
```

The help file gives details of all of the arguments for the function, whether they are required or optional and what their default values are.

Table 1-2 shows some useful mathematical functions.

Table 1-2: Useful Mathematical Functions

Purpose	Function
Exponential	`exp`
Natural logarithm	`log`
Log base 10	`log10`
Square root	`sqrt`
Cosine	`cos`
Sine	`sin`
Tangent	`tan`
Arc cosine	`acos`
Arc sine	`asin`
Arc tangent	`atan`
Round	`round`
Absolute value	`abs`
Factorial	`factorial`

Objects

In R, an *object* is some data that has been given a name and stored in the memory. The data could be anything from a single number to a whole table of data of mixed types.

Simple Objects

You can create objects with the *assignment operator*, which looks like this:

```
<-
```

For example, to create an object named `height` that holds the value `72.95` (a person's height in inches), use the command:

```
> height<-72.95
```

When creating new objects, you must choose an object name that:

- consists only of upper and lower case letters, numbers, underscores (_) and dots (.)
- begins with an upper- or lowercase letter or a dot (.)
- is not one of R's reserved words (enter `help(reserved)` to see a list of these)

R is case-sensitive, so `height`, `HEIGHT`, and `Height` are all distinct object names.

If you choose an object name that is already in use, you will overwrite the old object with the new one. R does not give any warning if you do this.

To view the contents of an object you have already created, enter the object name:

```
> height
```

```
[1] 72.95
```

Once you have created an object, you can use it in place of the information it contains. For example, as input to a function:

```
> log(height)
```

```
[1] 4.289774
```

As well as creating objects with specific values, you can save the output of a function or calculation directly to a new object. For example, the following command converts the value of the `height` object from inches to centimeters and saves the output to a new object called `heightcm`:

```
> heightcm<-round(height*2.54)
```

Notice that when you assign the output from a function or calculation to an object, R does not display the output. To see it, you must view the contents of the object by entering the object name.

To change the contents of an object, simply overwrite it with a new value:

```
> height<-69.45
```

Objects like these are called *numeric* objects because they contain numbers. You can also create other types of objects such as *character* objects, which contain a combination of any keyboard characters known as a *character string*. When creating a character object, enclose the character string in quotation marks:

```
> string1<-"Hello!"
```

You can use either double or single quotation marks to enclose a character string, as long they are both of the same type. To include quotation marks within a character string, place a backslash before the quotes (known as an *escape sequence):*

```
> string2<-"I said \"Hello!\""
```

So far we have only discussed simple objects that contain a single data value, but you can also create more complex types of objects. Two important types are *vectors* and *data frames*. A vector is an object that contains several data values of the same type. A data frame is an object that holds an entire dataset. Vectors and data frames are discussed in more detail in the following sections.

Vectors

A vector is an object that holds several data values of the same type arranged in a particular order. You can create vectors with a special function which is named `c`. For example, suppose that you have measured the temperature in degrees centigrade at five randomly selected locations and recorded the data as: 3, 3.76, -0.35, 1.2, -5. To save the data to a vector named `temperatures`, use the command:

```
> temperatures<-c(3, 3.76, -0.35, 1.2, -5)
```

You can view the contents of a vector by entering its name, as you would for any other object.

```
> temperatures
```

```
[1] 3.00 3.76 -0.35 1.20 -5.00
```

The number of values a vector holds is called its *length*. You can check the length of a vector with the `length` function:

```
> length(temperatures)
```

```
[1] 5
```

Each data value in the vector has a position within the vector, which you can refer to using square brackets. This is known as *bracket notation*. For example, you can view the third member of `temperatures` with the command:

```
> temperatures[3]
```

```
[1] -0.35
```

If you have a large vector (such that when displayed, the values of the vector fill several lines of the console window), the indices at the side tell you which member of the vector each line begins with. For example, the vector below contains twenty-seven values. The indices at the side show that the second line begins with the eleventh member and the third line begins with the twenty-first member. This helps you to determine the position of each value within the vector.

```
[1]   0.077  0.489  1.603  2.110  2.625  1.019  1.100  1.729  2.469 -0.125
[11]  1.931  0.155  0.572  1.160 -1.405  2.868  0.632 -1.714  2.615  0.714
[21]  0.979  1.768  1.429 -0.119  0.459  1.083 -0.270
```

If you give a vector as input to a function intended for use with a single number (such as the exp function), R applies the function to each member of the vector individually and gives another vector as output:

```
> exp(temperatures)
```

```
[1] 20.085536923 42.948425979  0.704688090  3.320116923  0.006737947
```

Some functions are designed specifically for use with vectors and use all members of the vector together to create a single value as output. An example is the `mean` function, which calculates the mean of all the values in the vector:

```
> mean(temperatures)
```

```
[1] 0.522
```

The `mean` function and other statistical summary functions are discussed in more detail in Chapter 5.

Like basic objects, vectors can hold different types of data values such as numbers or character strings. However, all members of the vector must be of the same type. If you attempt to create a vector containing both numbers and characters, R will convert any numeric values into characters. Character representations of numbers are treated as text and cannot be used in calculations.

Data Frames

A data frame is a type of object that is suitable for holding a dataset. A data frame is composed of several vectors of the same length, displayed vertically and arranged side by side. This forms a rectangular grid in which each column has a name and contains one vector. Although all of the values in one column of a data frame must be of the same type, different columns can hold different types of data (such as numbers or character strings). This makes them ideal for storing datasets, with each column holding a variable and each row an observation.

In Chapter 2, you will learn how to create new data frames to hold your own datasets. For now, there are some datasets included with R that you can experiment with. One of these is called `Puromycin`, which we will use here to demonstrate the idea of a data frame. You can view the contents of the `Puromycin` dataset in the same way as for any other object, by entering its name at the command prompt:

```
> Puromycin
```

R outputs the contents of the data frame:

```
      conc     rate         state
1     0.02       76       treated
2     0.02       47       treated
3     0.06       97       treated
4     0.06      107       treated
5     0.11      123       treated
6     0.11      139       treated
7     0.22      159       treated
8     0.22      152       treated
9     0.56      191       treated
10    0.56      201       treated
11    1.10      207       treated
12    1.10      200       treated
```

```
13    0.02       67     untreated
14    0.02       51     untreated
15    0.06       84     untreated
16    0.06       86     untreated
17    0.11       98     untreated
18    0.11      115     untreated
19    0.22      131     untreated
20    0.22      124     untreated
21    0.56      144     untreated
22    0.56      158     untreated
23    1.10      160     untreated
```

The dataset has three variables named `conc`, `rate`, and `state`, and it has 23 observations.

It is important to know how to refer to the different components of a data frame. To refer to a particular variable within a dataset by name, use the dollar symbol ($):

```
> Puromycin$rate
```

```
 [1]  76  47  97 107 123 139 159 152 191 201 207 200  67  51  84  86  98 115 131
[20] 124 144 158 160
```

This is useful because it allows you to apply functions to the variable, for example, to calculate the mean value of the `rate` variable:

```
> mean(Puromycin$rate)
```

```
[1] 126.8261
```

As well as selecting variables by name with the dollar symbol, you can refer to sections of the data frame using bracket notation. Bracket notation can be thought of as a coordinate system for the data frame. You provide the row number and column number between square brackets:

> *dataset[r,c]*

For example, to select the value in the sixth row of the second column of the `Puromycin` dataset using bracket notation, use the command:

```
> Puromycin[6,2]
```

```
[1] 139
```

You can select a whole row by leaving the column number blank. For example to select the sixth row of the `Puromycin` dataset:

```
> Puromycin[6,]
```

```
     conc   rate      state
6    0.11    139    treated
```

Similarly to select a whole column, leave the row number blank. For example, to select the second column of the `Puromycin` dataset:

```
> Puromycin[,2]
```

```
 [1]  76  47  97 107 123 139 159 152 191 201 207 200  67  51  84  86  98 115 131
[20] 124 144 158 160
```

When selecting whole columns, you can also leave out the comma entirely and just give the column number.

```
> Puromycin[2]
```

```
      rate
1       76
2       47
3       97
4      107
5      123
6      139
```

```
7      159
8      152
9      191
10     201
11     207
12     200
13      67
14      51
15      84
16      86
17      98
18     115
19     131
20     124
21     144
22     158
23     160
```

Notice that the command `Puromycin[2]` produces a data frame with one column, while the command `Puromycin[,2]` produces a vector.

You can use the minus sign to exclude a part of the data frame instead of selecting it. For example, to exclude the first column:

```
> Puromycin[-1]
```

You can use the colon (:) to select a range of rows or columns. For example, to select row numbers six to ten:

```
> Puromycin[6:10,]
```

```
       conc    rate          state
6      0.11     139        treated
7      0.22     159        treated
8      0.22     152        treated
9      0.56     191        treated
10     0.56     201        treated
```

To select nonconsecutive rows or columns, use the c function inside the brackets. For example, to select columns one and three:

```
> Puromycin[,c(1,3)]
```

```
       conc          state
1      0.02        treated
2      0.02        treated
3      0.06        treated
4      0.06        treated
5      0.11        treated
6      0.11        treated
7      0.22        treated
8      0.22        treated
9      0.56        treated
10     0.56        treated
11     1.10        treated
12     1.10        treated
13     0.02      untreated
14     0.02      untreated
15     0.06      untreated
16     0.06      untreated
17     0.11      untreated
18     0.11      untreated
19     0.22      untreated
20     0.22      untreated
21     0.56      untreated
22     0.56      untreated
23     1.10      untreated
```

You can also use object names in place of numbers:

```
> rownum<-c(6,8,14)
```

```
> colnum<-2
> Puromycin[rownum,colnum]
```

```
[1] 139 152  51
```

Or even functions:

```
> Puromycin[sqrt(25),]
```

```
      conc    rate          state
5     0.11     123        treated
```

Finally, you can refer to specific entries using a combination of the variable name and bracket notation. For example, to select the tenth observation for the `rate` variable:

```
> Puromycin$rate[10]
```

```
[1] 201
```

You can view more information about the `Puromycin` dataset (or any of the other dataset included with R) with the `help` function:

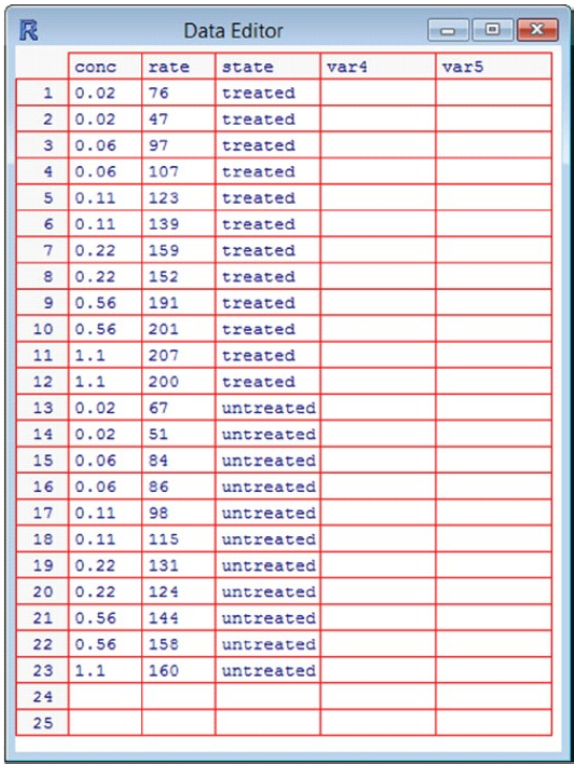
```
> help(Puromycin)
```

The Data Editor

As an alternative to viewing datasets in the command window, R has a spreadsheet style viewer called the *data editor*, which allows you to view and edit data frames. To open the `Puromycin` dataset in the data editor window, use the command:

```
> fix(Puromycin)
```

Alternatively, you can select Data Editor from the Edit menu and enter the name of the dataset that you want to view when prompted. The dataset opens in the data editor window, as shown in Figure 1-2. Here you can make changes to the data. When you have finished, close the editor window to apply them.



	conc	rate	state	var4	var5
1	0.02	76	treated		
2	0.02	47	treated		
3	0.06	97	treated		
4	0.06	107	treated		
5	0.11	123	treated		
6	0.11	139	treated		
7	0.22	159	treated		
8	0.22	152	treated		
9	0.56	191	treated		
10	0.56	201	treated		
11	1.1	207	treated		
12	1.1	200	treated		
13	0.02	67	untreated		
14	0.02	51	untreated		
15	0.06	84	untreated		
16	0.06	86	untreated		
17	0.11	98	untreated		
18	0.11	115	untreated		
19	0.22	131	untreated		
20	0.22	124	untreated		
21	0.56	144	untreated		
22	0.56	158	untreated		
23	1.1	160	untreated		
24					
25					

Figure 1-2: The data editor window

Although the data editor can be useful for making minor changes, there are usually more efficient ways of manipulating a dataset. These are covered in Chapter 3.

Workspaces

The *workspace* is the virtual area containing all of the objects you have created in the session. To see a list of all of the objects in the workspace, use the `objects` function:

```
> objects()
```

You can delete objects from the workspace with the `rm` function:

```
> rm(height, string1, string2)
```

To delete all of the objects in the workspace, use the command:

```
> rm(list=objects())
```

You can save the contents of the workspace to a file, which allows you to resume working with them at another time.

Windows users can save the workspace by selecting File then Save Workspace from the drop-down menus, then naming and saving the file in the usual way. Ensure that the file name has the `.RData` file name extension, as it will not be added automatically.

R automatically loads the most recently saved workspace at the beginning of each new session. You can also open a previously saved workspace by selecting File, then Open Workspace, from the drop-down menus and selecting the file in the usual way. Once you have opened a workspace, all of the objects within it are available for you to use.

Mac users can find options for saving and loading the workspace from the Workspace menu.

Linux users can save the workspace by entering the command:

```
> save.image("/home/Username/folder/filename.RData")
```

The file path can be either absolute or relative to the home directory.

To load a workspace, use the command:

```
> load("/home/Username/folder/filename.RData")
```

Error Messages

Sometimes R will encounter a problem while trying to complete one of your commands. When this happens, a message is displayed in the console window to inform you of the problem. These messages come in two varieties, known as *error messages* and *warning messages*.

Error messages begin with the text `Error:` and are displayed when R is not able to perform the command at all. One of most common causes of error messages is giving a command that is not a valid R command because it contains a symbol that R does not understand, or because a symbol is missing or in the wrong place. This is known as a *syntax error*. In the following example, the error is caused by an extra closing parenthesis at the end of the command:

```
> round(3.141592))
```

```
Error: unexpected ')' in "round(3.141592))"
```

Another common cause of errors is mistyping an object name so that you are referring to an object that does not exist. Remember that object names are case-sensitive:

```
> log(object5)
```

```
Error: object 'object5' not found
```

The same applies to function names, which are also case-sensitive:

```
> Log(3.141592)
```

```
Error: could not find function "Log"
```

A third common cause of errors is giving the wrong type of input to a function, such as a data frame where a vector is expected, or a character string where a number is expected:

```
> log("Hello!")
```

```
Error in log("Hello!") : Non-numeric argument to mathematical function
```

Warning messages begin with the text `Warning`: and tell you about issues that have not prevented the command from being completed, but that you should be aware of. For example, the command below calculates the natural logarithm of each of the values in the `temperatures` vector. However, the logarithm cannot be calculated for all of the values, as some of them are negative:

```
> log(temperatures)
```

```
[1] 1.0986123 1.3244190       NaN 0.1823216       NaN
Warning message:
In log(temperatures) : NaNs produced
```

Although R is still able to perform the command and produce output, it displays a warning message to draw to your attention to this issue.

Script Files

A script file is a type of text file that allows you to save your commands so that they can be easily reviewed, edited, and repeated.

To create a new script file, select New Script from the File menu. Mac users should select New Document from the File menu. This opens a new R Editor window where you can type commands. The Linux version of R does not include a script editor; however, a number of external editors are available. To see a list of these, go to www.sciviews.org/_rgui/projects/Editors.html.

To run a command from the R Editor in the Windows environment, place the cursor on the line that you want to run, then right-click and select Run Line or Selection. You can also use the shortcut Ctrl+R. Alternatively, you can click the run button, which looks like this:

To run several commands, highlight a selection of commands then right-click and select Run Line or Selection, as shown in Figure 1-3.

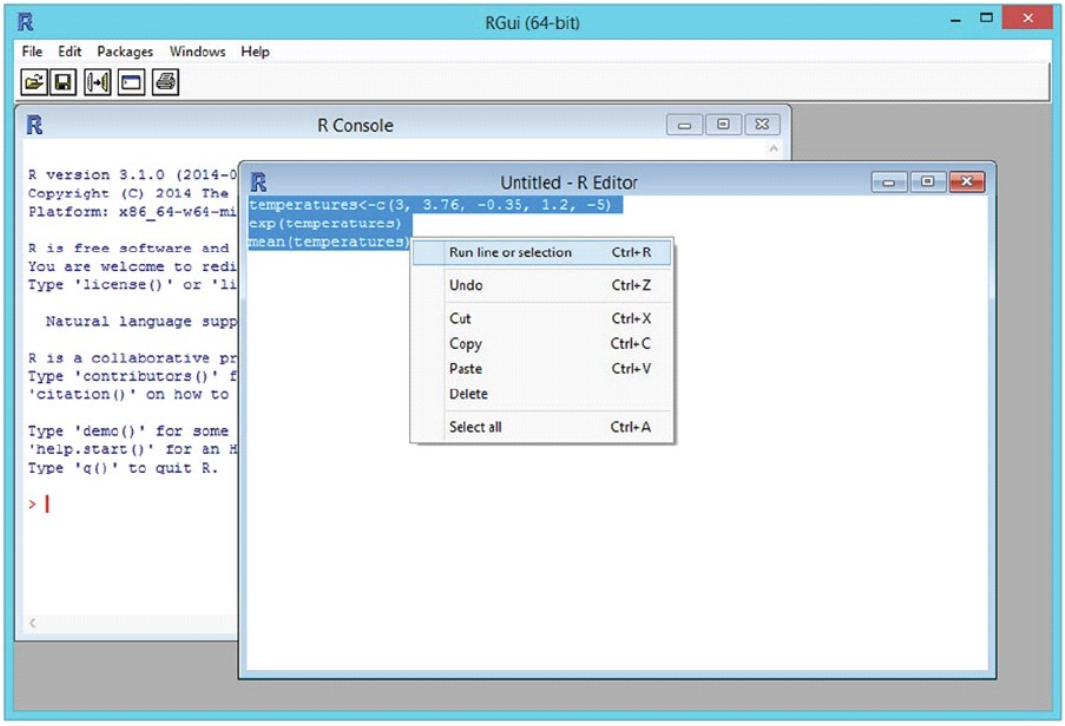


Figure 1-3: Running commands from a script file in the Windows environment

Mac users can run the current line or a selection of commands by pressing Cmd+Return.

Once you have run the selected commands, they are submitted to the command window and executed one after the other.

If your script file is going to be used by someone else or if you are likely to return to it after a long time, it is helpful to add some *comments*. Comments are additional text that are not part of the commands themselves but are used to make notes and explain the commands.

Add comments to your script file by typing the hash sign (#) before the comment. R ignores any text following a hash sign for the remainder of the line. This means that if you run a section of commands that has comments in, the comments will not cause errors. Figure 1-4 shows a script

file with comments.

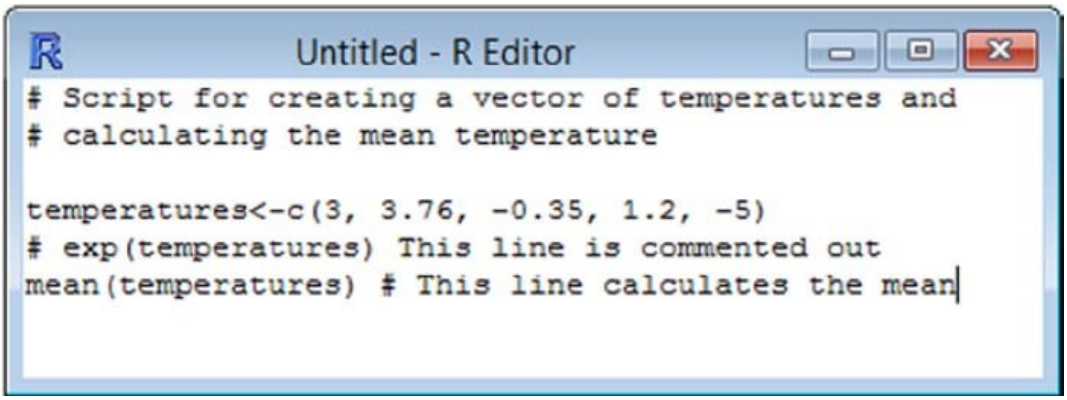


Figure 1-4: Script file with comments

You can save a script file using by selecting File, then Save. If the Save option is not shown in the File menu, it is because you don't have focus on the script editor window and need to select it. The file is given the *.R* file name extension. Similarly, you can open a previously saved script file by selecting File, then Open Script, and selecting the file in the usual manner.

Mac users can save a script file by selecting the icon in the top left-hand corner of the script editor window. They can open a previously saved script file by selecting Open Document from the File menu.

Summary

The purpose of this chapter is to familiarize you with the R interface and the programming terms that will be used throughout the book. Make sure that you understand the following terms before proceeding:

- **R Console** The window into which you type your commands and in which output and any error or warning messages are displayed.
- **Command** A typed instruction to R.
- **Command prompt** The symbol used by R to indicate that it is ready to receive your command, which looks like this: >.
- **Function** A set of commands that have been given a name and together perform a specific task.
- **Argument** A value or piece of data supplied to a function as input.
- **Object** A piece of data or information that has been stored and given a name.
- **Vector** An object that contains several data values of the same type arranged in a particular order.
- **Data frame** A type of object that is suitable for holding a dataset.
- **Workspace** The virtual area containing all of the objects created in the session, which can be saved to a file with the `.RData` file name extension.
- **Script file** A file with the `.R` extension, which is used to save commands and comments.

Now that you are familiar with the R interface, we can move on to Chapter 2 where you will learn how to get your data into R.