Introduction to R for data scientists

Dr. Pranshu Gupta, Dr. Ramon A. Mata-Toledo, Dr. Ana Stanescu

December 2018

Contents

1 Basic Introduction to the R Language 2

1.1 What is R? 2
1.2 Features of R 2
1.3 R vs. other programming languages 2
1.4 Why use R for data science? 3
1.5 Version 3
1.6 R packages 3
1.7 Introduction to RStudio 3
1.7.1 RStudio session 4
1.7.2 Working Directory 4
1.7.3 Ending a session 4
1.8 Comments in R 5
1.9 Basic Arithmetic Operations in R 5
1.9.1 Priority of Operations 6
1.9.2 Incomplete Commands in R 7
1.10 Variables in R 8
1.11 How are Variables Named in R? 8
1.12 Reserved or Keywords in R 8
1.13 Assigning Values to Variables 9
1.14 Removing Variables in a Session 10
1.15 Numeric and Mixed Types in R 11
1.15.1 Integer Division 11
1.16 Rounding Numbers in R 12
1.16.1 The Ceiling(), Floor(), and Trunc() functions 14
1.17 Character Data 15
1.18 Summary 16

Chapter 1

Basic Introduction to the R Language

In this chapter, we consider the fundamental aspects and characteristics of the programming language R. Chapters 1 through 3 will serve as an introduction to R with the aim to get familiar with R's data structures and apply them to the field of data science. Additional information can be obtained directly from the Comprehensive R Archive Network (CRAN) at

`htpp://cran.r-project.org/doc/html/interface98-paper/paper.html`

1.1 What is R?

- R is a full programming language used in a variety of environments such as banking, advertisement, insurance, and medicine among other fields. It is widely used for data analytics, big data, and data sciences.
- R is a language and environment developed by Ross Ihaka and Robert Gentleman at the University of Auckland (New Zealand). R is an extension of another statistical program called S developed at Bell Laboratories during the mid to late 70s. To learn more about the history of R, visit the CRAN website.
- R is free and made available under the Free Software Foundation's GNU General Public License (GPL) version 2. Under this license, R code can be changed but required to be redistributed after the changes are complete.
- R can be downloaded from the URL previously shown. This website contains additional information about the language R including manuals, mailing lists, and some ancillary material.

1.2 Features of R

Although there are many advantages and features that are extremely useful in R, a few are listed below:

- R runs on all platforms Windows, Mac, and all variations and descendants of UNIX and Linux.
- R suite includes operators for calculations on arrays and especially matrices.
- It is an effective programming language that includes graphical facilities for data analysis.
- It can be extended by means of add-packages of Q & A Questions such as:

`www.stackoverlflow.com/question/tagged/r`
`www.stats.stackexchange.co./questions/tagged/r`

1.3 R vs. other programming languages

- R is a vector language, meaning that an object variable can hold more than one value at a time. A vector is a collection of elements of the same type.
- R allows operations on vectors without the explicit need of loops or any other iterative constructs.
- R provides, through the use of packages, all resources needed for performing data analysis.
- R can interact with all databases and most programming languages. This allows extending the capabilities of the language.
- R contains many statistical routines not available in other statistical programs due to the fact that new methods are uploaded frequently.

1.4 Why use R for data science?

- Data Wrangling - R has an extensive library of tools for database manipulation and wrangling
- Data Visualization - R has many tools that can help in data visualization, analysis, and representation.
- Machine Learning - R provides tools to train and evaluate an algorithm and predict future events.

1.5 Version

There are a few interfaces available for using R such as

- Windows - RGui
- Mac OS X - R.app
- Linux - packages available for Redhat, Ubuntu and more

In this book, a third-party interface known as **RStudio** that is supported by all three platforms will be used. In addition to RStudio, there are some other integrated development environments (IDEs) that can be used for statistical, and data mining tasks are Eclipse StatET, Emacs Speaks Statistics and Rattle. RStudio includes the following:

- A smart editor that allows scripting, i.e., writing code in a file and executing the file to display output.
- A console for writing interactive code that is executed as the code is being typed.
- Environment and History tabs (where all "local" or "global" variables created during a session are saved).
- Tabs for Files, Plots, Packages, Help and a Viewer pane provides an area to browse folders and files, display plots (charts or graphs), and view a list of installed packages.

All R commands exhibit the same behavior in Windows, Mac or Linux environments. The corresponding desktop version of RStudio for Windows, Mac, or Linux can be downloaded free from the following website:

`https://www.rstudio.com`.

1.6 R packages

A package is a pre-written piece of code specifically designed to accomplish a task. R is simple, effective and popular due to use of a large collection of well-defined and well-maintained packages. All R packages are available on the CRAN website. The effectiveness of all package may not be the same.

Every package available to R users cannot be included in the book due to space restrictions; therefore, the widely used and highly reviewed packages will be demonstrated in this book. Some packages will be introduced briefly in this book.

1.7 Introduction to RStudio

The interface of RStudio is divided into four panes: Source (writing scripts), Console/Terminal (interaction with R), Environment/History/Connections (show variables created during the session), and Files/Plots/Packages/HelpViewer. The latter pane displays plots and graphs shows all packages available to the session and displays the help information requested by the user.

When R is launched the first time, the Source pane may not show up. To see all four panes, perform the following commands, in sequence, using the RStudio menu:

View → Panes → Show all panes

1.7.1 RStudio session

Once installed, click on the R logo that appears on the desktop to start R. R can be stopped by clicking "Ctrl + Q." RStudio also allows stopping the R session using the menu. RStudio offers different ways of saving the work such as:

1. The individual variables can be saved, one at a time, using the function **save(x)** where "x" stands for the individual variable to be saved.
2. The environment can be saved by using the function **save.image()**. This function is actually a shortcut to save the "current environment". The environment is saved in the current working directory (See section Working with the Current Directory."
3. The R script can be saved by clicking on the save logo of the Script pane , or using the R menu click the sequence:

   Session → Save workspace as

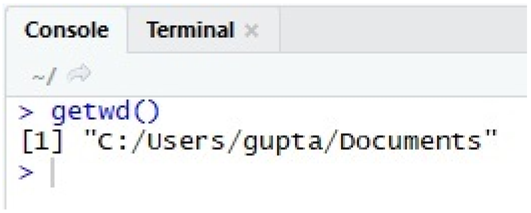
4. The two floppies icon in the R menu allows saving all documents at once.

1.7.2 Working Directory

By default, every time a file is read or written to, RStudio uses the current directory. Generally, RStudio, during its installation, will ask the user the desired location for the working directory. The default setting can be used in this scenario or can be changed to another location. To find the location of the current directory, use the R function **getwd()**. Notice that this function does not have an argument. The following examples are tested using a Mac computer. See Note 1.1 when using a Windows computer.

Example No. 1.1 Getting the Working Directory

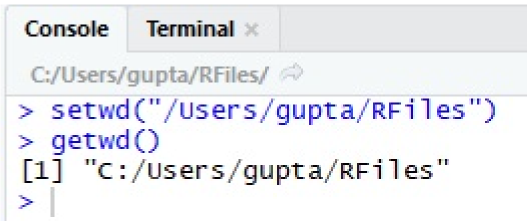
The output of the function getwd() at the Console prompt will display the current working directory as shown below:



```
> getwd()
[1] "C:/Users/gupta/Documents"
>
```

Example No. 1.2 Changing the Current Working Directory

Assuming a folder called R-folder is already created. The current working directory can be changed using the following command:



```
> setwd("/Users/gupta/RFiles")
> getwd()
[1] "C:/Users/gupta/RFiles"
>
```

Notice that it is necessary to write the working directory inside quotes.

Note No. 1.1 As far as writing the path to a file in a Mac or a Windows machine, we need to be aware of the type of slash being used. If we are working with Windows, you can use the frontward slash. However, this may look weird because we may be accustomed to using the backward slash ("\"). However, because this slash is also used as the escape character in Windows, it is necessary to double it. For example, in a Windows machine, the working directory can be set as indicated in the next example.

1.7.3 Ending a session

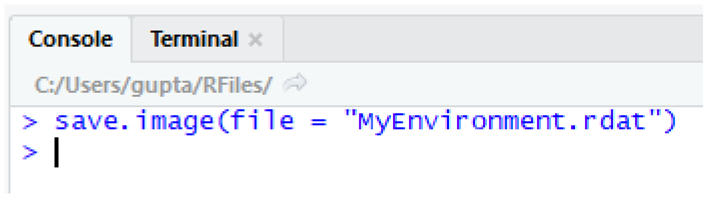
A session can be ended in R using one of the following actions:

- Save the R script using the commands "File -> Save" or "File -> Save as" from the File menu.

- Save all objects created during the session (The Environment) using the **save.image()** function. The file name should be enclosed in quotes. Although it is not necessary, the extension of this file is ".rdat" which serves as a mnemonic to remind us that this is a file with "r data."
- Save selected variables with the **save()**function.
- Save the history of all commands used in your session using the **savehistory()** function.

The following example illustrates these functions.

Example No. 1.3 Saving and Recovering the environment at the end of a session



```
> save.image(file = "MyEnvironment.rdat")
> 
```

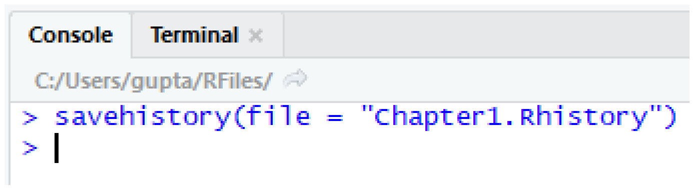
To illustrate how to recover the environment, we use the clear "all objects" in the Environment pane by clicking on the R broom logo .

Next, we can use the **load()** function as follows:

load("MyEnvironment")

The same effect can be obtained after quitting a session using the function **quit(save='no')** and then loading the environment again. The latter commands obviously assume that you have already saved the environment before quitting with the 'no' option.

Example No. 1.4 Saving the History of all commands of a Session



```
> savehistory(file = "Chapter1.Rhistory")
> 
```

By default, R saves the history file under the current directory as an ASCII file which can be opened with any text editor. The extension of the file is nothing more than a reminder to the user that the commands used in a particular file are R commands. Using the command shown above, the file name "Chapter1.Rhistory" will contain all the R's commands used in the creation of the file named "Chapter1."

1.8 Comments in R

In the Console pane, all comments are initiated with the hash or pound symbol #. Comments can appear anywhere in a line, but once R recognizes the # symbol, it ignores anything after it on that line.

There is no way to create multi-line comments in R in the Console pane. There can be multiple lines of comments but each line must begin with a # sign.

In the Script pane, highlight the lines to comment and then use the following options from the RStudio menu:

Code $\longrightarrow$ Comment

1.9 Basic Arithmetic Operations in R

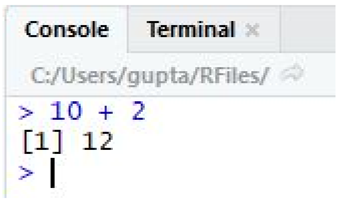
When working with R in Console, R uses the prompt > to show it is expecting a user command. After this prompt, we can type any arithmetic or algebraic operation. In this case, R, is nothing more than a glorified calculator. The basic mathematical operators are shown in the following table:

Operator	Symbol	Example	Result
Addition	+	10+2	12
Subtraction	−	5−7	−2
Multiplication	∗	3∗2	6
Division	/	10/2	5
Exponentiation	∧	2∧3	8
Modulus	%%	15%%2	1(remainder)
Floor Operator	%/%	15%/%2	7

Table 1. Arithmetic Operators

Example No. 1.5 Basic arithmetic operations in R

Let's consider the following addition:



```
> 10 + 2
[1] 12
> 
```

Line [1] shows the output of R which is the addition of two numbers. R always answers with a number enclosed in square brackets at the beginning of the line. More explanation about this will come in future chapters. However, what R is telling us is that the number or character that follows [1] is the first answer to the result. Because this addition produces only one result we will see as the answer. In cases that the answer of operation exceeds the capacity of one written line, the number enclosed inside the square brackets at the beginning of every line indicates the position that the first number, to the right of the brackets, occupies in the result of the operation. For more explanation, refer to Example 4 of Chapter 2.

When some mathematical operations are combined we will assume that the operations, unless parentheses are used, will be carried out according to the priority indicated in the table shown below. Note: The operations are carried out from left to right unless parenthesis or the hierarchy of operators change this sequence.

1.9.1 Priority of Operations

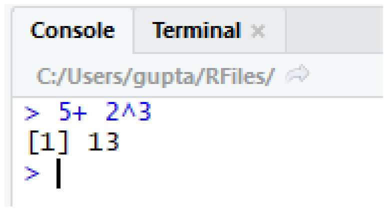
Exponentiation	Highest
Multiplication\Division	High
Addition\Subtraction	Lowest

Table 2. Priority of Arithmetic Operators

As a mnemonic for the use of the priority of operators, we can use the **PEMDAS** acronym which establishes the order in which arithmetics operations are carried out. Here **P** stands for parenthesis, **E** for exponential, **M** for multiplication, **D** for the division, **A** for addition, and finally, **S** for subtraction. This mnemonic reminds us that operations inside parenthesis are performed first. That is, they take priority over any other operation. In case there are operations in parentheses within other parentheses, these operations will be carried out starting from the innermost to the outermost parentheses. Then any exponentiation operation is performed next. Operations which have the same priority (multiplication and division) or (addition and subtraction) are carried out from left to right unless the priority is altered by the use of parentheses.

Example No. 1.6 Applying the priorities of the operators to a mathematical expression

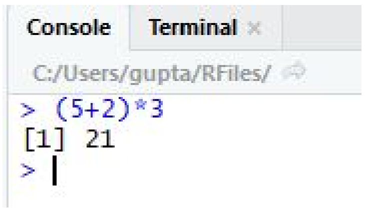
The result of the operation $5+2^3$ is equal to 13. Notice that the exponentiation will be performed first because its priority is higher than that of the addition. This exponentiation alters the ordinary left-to-right order of execution. At the R console, we will write this as follows:



```
> 5+ 2^3
[1] 13
>
```

The output shows the calculated value of the expression. **Example No. 1.7 Using parentheses to override the priority of the operators**

The result of the operation $(5+2)*3$ is equal to 21. Notice that the expression in parenthesis will be performed first because it has the highest priority. Therefore, the addition of $5+2$ or 7 will be carried out or performed first because its priority is higher than that of addition. The result of this operation is 21 as should it be expected. At the R console, we will write this as follows:

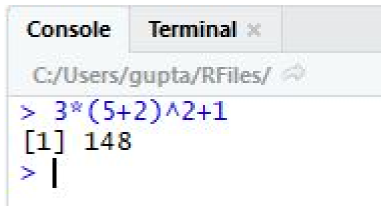


```
> (5+2)*3
[1] 21
>
```

The output shows the value of the expression where the parenthesis is evaluated first followed by the multiplication.

Example No. 1.8 Another example to show the priority of operators

The result of the operation $3*(5+2)^2+1$ is equal to 148. Notice that the expression in parenthesis will be performed first because it has the highest priority. Therefore, the addition of $5+2$ or 7 will be carried first. The exponentiation will be performed second. That is, 7 will be squared. This will produce 49. Then this result will be multiplied by 3 producing 147. Finally, the 1 will be added to produce the final result of 148. At the R console, we will write this as follows:



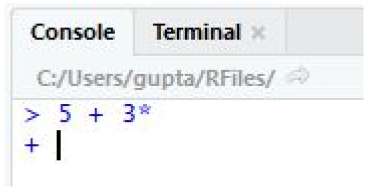
```
> 3*(5+2)^2+1
[1] 148
>
```

The calculated value of the complex operation with multiple operators is shown above.

1.9.2 Incomplete Commands in R

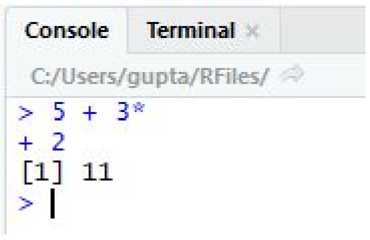
On occasions, the user may press the Enter key before finishing an arithmetic operation or any other valid R command; this may also happen when the operations are too long and do not fit on a single line. In cases like these, R will indicate that something is amiss by writing a "+" sign where the usual R prompt should appear. The following example will illustrate this.

Example No. 1.9
When performing a calculation such as 5 + 3*2, if the user presses the Enter key accidentally after the "*" then R will write a "+" sign to indicate that the operation is not complete.



```
> 5 + 3*
+
```

Once the expression is completed by typing the number 2, R will execute the command and will show the following result:



```
> 5 + 3*
+ 2
[1] 11
> 
```

1.10 Variables in R

Unlike strong-typed languages such as Java or C++ that check the type of each variable and how and where they are used, the language R does not require that variables be associated with any particular type. Variables in R can be of many types such as numeric, character, date (some of which may be time-based or stamped), and logical (Boolean).

In R, any name where a data can be stored and retrieved is called an **object.** Therefore, variables are objects which can hold values of other objects such as functions, a plot, a graph or the result of an analysis.

Variables in R can hold, at any time, one and only any of the previous types. The user can assign to a variable a numeric value and, later on, assign to the same variable an object of different type. However, once a variable has been assigned a value, it will retain this value until a new value is assigned to it.

1.11 How are Variables Named in R?

Variables in R can be formed by combining alphanumerics (letters and numbers) including periods ("."), dashes ("-"), or underscores ("_"). *However, variables cannot begin with a number.* Although it is possible to begin a variable with a "period" but it cannot be followed by a digit. Therefore, it is recommended that you do not start variables names with "periods" or "dots". Also, keep in mind that variables in R are **case-sensitive.** That is, R considers lowercase and upper case letters as different. For example, a variable named ExpectedValue is different than any other variable named expectedValue or Expectedvalue. Notice that the variables may read the same but they will be considered different in R.

There is no single convention for writing variables in R. However, in the field of computer science there are many conventions that have been proposed for writing variables. In this book, the authors suggest the "CamelCase" notation where compound variable names (made up of more than one word) begin with a capital letter and each word in the middle begins also with a capital letter. The variable ExpectedValue is an example of this convention. CamelCase notation is widely used in for brand names such iPhone, US post office, etc.

For some examples in this book, a single-letter variables names (x, y, s, etc.) is used to illustrate examples such as how a particular function or command works. However, in later sections, the book makes extensive use of the CamelCase notation.

1.12 Reserved or Keywords in R

Although there is absolute freedom in choosing variable names, the notion of "Free Speech" does not apply in R. Therefore, there are words that cannot be used as object names. The list shown below indicates some of these reserved or keywords. Because R is open-software, there may be additional reserved words in the future. The following list shows some of the currently reserved words.

Keyword Words
FALSE
TRUE
if
Inf
else
while
repeat
return
next
break
for
NULL
function
NaN
NA
seq

Table 3. Reserved Keywords

Note No. 1.2 The authors recommend not to use T or F as abbreviations of TRUE and FALSE. There is a high probability that your program will fail due to internal conflict with the variables used in some packages or the internal operations of R.

1.13 Assigning Values to Variables

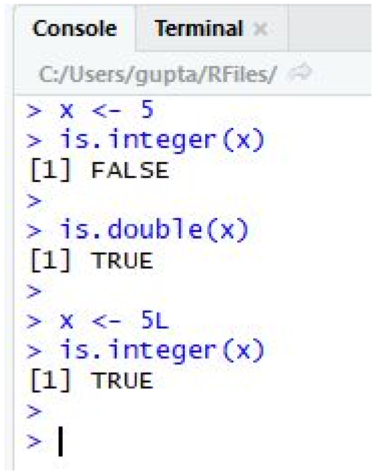
R has many ways of assigning values to objects. The preferable operator is "<-" (a greater sign symbol followed by a hyphen with no space between them. Do not write the quotes.) This operator assigns a value to an object. The general format of an assignment command is as follows:

object <- value

The value assigned to a variable can be a numeric integer, double, or character. Variables of type double are also called float in some other languages. Character-string variables need to be enclosed in quotes, single or double. Our suggestion is that you choose one style, single or double, and be consistent with using one of the styles. We need to keep in mind that for R, the word integer refers to the way the value is stored internally in memory and not how we think of the value itself. To make sure that R treats a variable as an integer, we need to append the letter "L" right after the number. In the following example, we will make use of the R function **is.integer(x)** which returns a TRUE value if the parameter "x" is a variable which has been assigned a whole number with an L at the end integer. However, this function, should not be used to test if any given result is an integer value. The TRUE or FALSE value as a result of using **is.integer(x)** does not refer to the value x itself but how this variable is stored internally by R. After assigning a value to a variable, if we type just the name of the variable at the console prompt we can see its content. The content will be displayed on the next line of the console. If the variable is numeric, R will show only the value and will not indicate its value. To see the type of the variable we can use some R functions as illustrated in the next examples to determine if a variable is of type integer or not. Another way of knowing the type of a variable is to look at the Environment pane located, by default, in the upper-right corner of the RStudio interface.

Example No. 1.10 Assigning an integer value to a variable

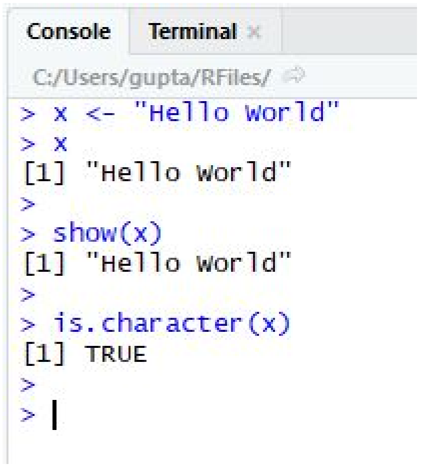
Assuming a variable name "x", we can assign a value to this variable as x <- 5 or x <- 5L. The Illustration below shows the results of using the "is.integer(x)" function after the different assignment commands.



```
> x <- 5
> is.integer(x)
[1] FALSE
>
> is.double(x)
[1] TRUE
>
> x <- 5L
> is.integer(x)
[1] TRUE
>
> 
```

With no L at the end of the number, the numeric value is stored as a double. Adding the L at the end of the number stores the number as an integer.

Example No. 1.11 Assigning a string to a variable



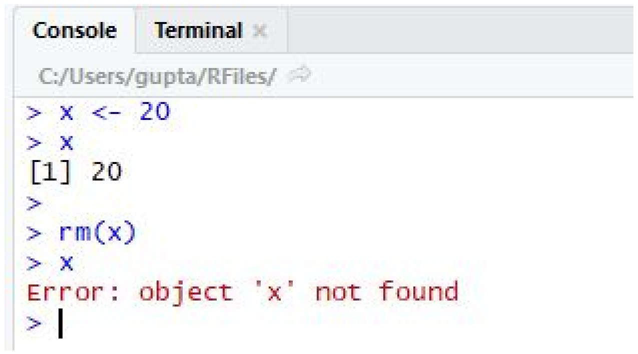
```
> x <- "Hello World"
> x
[1] "Hello World"
>
> show(x)
[1] "Hello World"
>
> is.character(x)
[1] TRUE
>
> 
```

The above example shows two options to view the value of a variable. The name of the variable can be used to retrieve the value or the function show(x) can be used where x is the name of the variable. The example shown above also checks if x is a character value.

1.14 Removing Variables in a Session

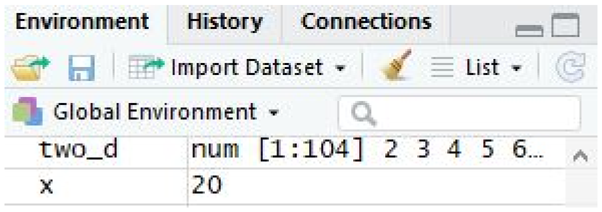
There may be reasons why a variable may no longer be necessary or needs to be eliminated from analysis. To eliminate a variable, the remove function **rm(x)** can be used where x is the variable to be removed.

Example No. 1.12 Removing a Variable from a Session



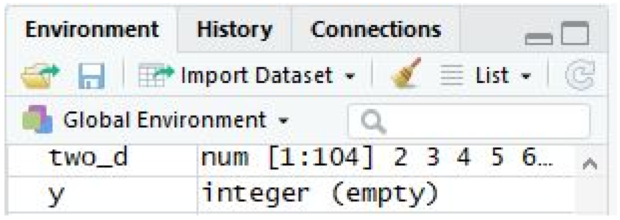
```
> x <- 20
> x
[1] 20
>
> rm(x)
> x
Error: object 'x' not found
> 
```

It can be seen from the above example that once the variable is removed the contents of the variable cannot be retrieved. The Environment pane in Rstudio provides an overview of all the variables declared, once a variable has been removed it will not be listed in the Environment pane. When the variable is created, it is added to the Environment pane as shown below:



```
two_d    num [1:104] 2 3 4 5 6...
x        20
```

When the variable is removed it is not visible in the Environment pane:

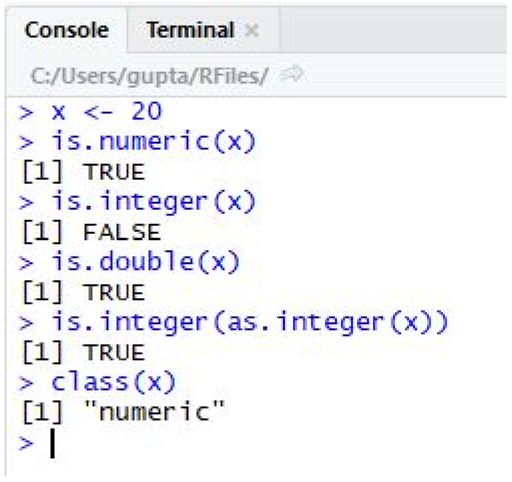


```
two_d    num [1:104] 2 3 4 5 6...
y        integer (empty)
```

1.15 Numeric and Mixed Types in R

As we indicated before, R assumes that all variables are of type double unless indicated otherwise. If variables of type float and integer variables are used in a mathematical expression, R will promote all variables and results to double by default. To determine if a variable "x" is double we can use the function **is.double(x).** To determine if a variable "x" is numeric we can use the function **is.numeric(x).** The function **is.numeric(x)** will work with both integer and double but it will only indicate if the variable "x" is numeric not if it is double or integer. The function **as.integer(x)**attempts to coerce the variable "x" to type integer. The answer will be NA unless the coercion succeeds. NA stands for "not available" or "missing values." R also has another function called **class(x)** that can help determine if the type of a variable is numeric or not. If the parameter is a value followed by the letter "L" it will return the value "integer" otherwise, it will return the value "numeric". In this case, "numeric" means "double." The following example will illustrate this.

Example No. 1.13 Using the is.numeric(x) and is.double(x) function

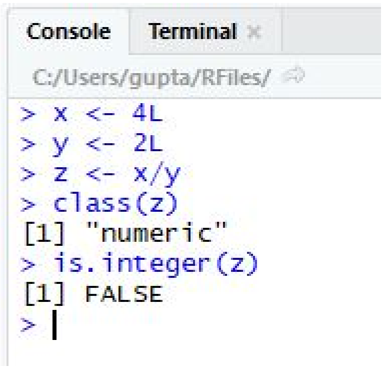


```
> x <- 20
> is.numeric(x)
[1] TRUE
> is.integer(x)
[1] FALSE
> is.double(x)
[1] TRUE
> is.integer(as.integer(x))
[1] TRUE
> class(x)
[1] "numeric"
>
```

1.15.1 Integer Division

Whenever integer and double values are used in an arithmetic expression, R will "promote" to double the result. In case of dividing two integer numbers, even if the dividend is multiple of the divisor, the result will be "promoted" to double. This makes sense since the division of two integers is not necessarily an integer. As the mathematician will say the "integer division does not satisfy the closure property." That is, the result of dividing two integers does not have to be another integer.

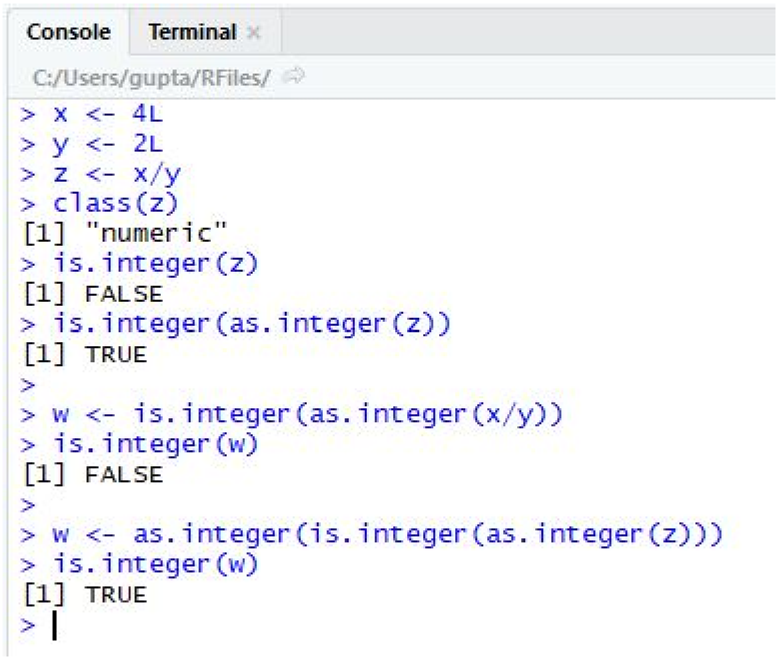
Example No. 1.14 Dividing two integer variables



```
> x <- 4L
> y <- 2L
> z <- x/y
> class(z)
[1] "numeric"
> is.integer(z)
[1] FALSE
> 
```

The value of x is multiple of the variable y. The result of the division is of type double and not integer. As indicate before, the function **"as.integer"** attempts to coerce its argument to type integer. If the coercion fails, R will produce an NA. Sometimes R produces the error message "NaN" which stands for "Not a number." It is important to notice these two types of error messages, although may look similar, do not mean the same. NA may come in different flavors such as described on the following website: `www.r-bloggers.com/difference-between-na-and-nan-in-r`.

Example No. 1.15 Coercing an argument to an integer value // As it is illustrated in the following example.



```
> x <- 4L
> y <- 2L
> z <- x/y
> class(z)
[1] "numeric"
> is.integer(z)
[1] FALSE
> is.integer(as.integer(z))
[1] TRUE
>
> w <- is.integer(as.integer(x/y))
> is.integer(w)
[1] FALSE
>
> w <- as.integer(is.integer(as.integer(z)))
> is.integer(w)
[1] TRUE
> 
```

1.16 Rounding Numbers in R

As already mentioned, unless otherwise indicated, R assumes that the numerical values are of type double. When doing mathematical operations R can handle up to 16 decimal digits. Most of the times it is not necessary to work with that many digits. R has a variety of functions that let us work with a smaller number of decimal digits. The use of these functions is illustrated below.

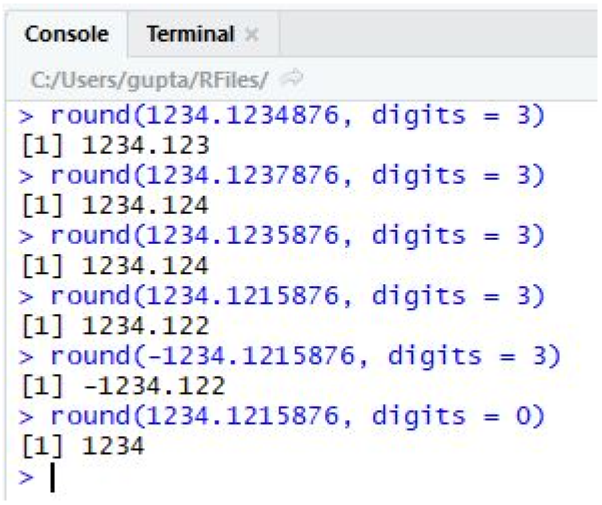
1. round(x)
2. signif(x)

3. floor(x)

4. ceiling(x)

5. trunc(x)

Example No. 1.16 Using the function round()



Console Terminal

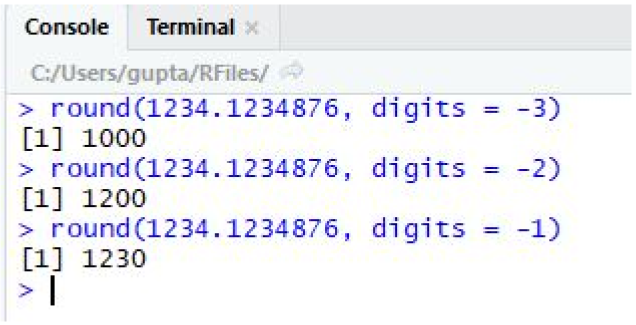
C:/Users/gupta/RFiles/

```
> round(1234.1234876, digits = 3)
[1] 1234.123
> round(1234.1237876, digits = 3)
[1] 1234.124
> round(1234.1235876, digits = 3)
[1] 1234.124
> round(1234.1215876, digits = 3)
[1] 1234.122
> round(-1234.1215876, digits = 3)
[1] -1234.122
> round(1234.1215876, digits = 0)
[1] 1234
> 
```

In this example, observe that the **round()** function has been used with four slightly different values all of them with 7 decimal digits, however, in all cases, only 3 digits are retained. Hence, the use of the option **"digits = 3"** . In the first case, R rounded the value to 1234.123 but in the second case, it rounded to 1234.124. The difference between these two results has to do with the "fourth" decimal digit and whether this digit is greater or equal to 5. Observe that in the first case, the fourth digit is "4" whereas in the second case is "7." Since we only want to retain 3 digits, R will take into account the fourth digit. In the first case, the fourth digit is less than 5. Therefore, R it will keep the third digit "as is." That is the reason why R produced the result 1234.123. In the second case, the fourth decimal digit is "7." Since this fourth digit is greater than 5, then R will increment the third digit, 3, to the next integer value by adding 1, in this case, 4. As a result the number 1234.124 was obtained. What about if the fourth value is exactly 5? In these cases, R increments the third digit by 1. That is why in the third and fourth cases respectively, R produces the results 1234.124 and 1234.122.

This rule is also true for negative numbers as shown in the fifth case. If the option "digits = 0" is used all the decimal digits are dropped as seen in the sixth case.

Example No. 1.17 Rounding to multiples of 10 using a negative argument



Console Terminal

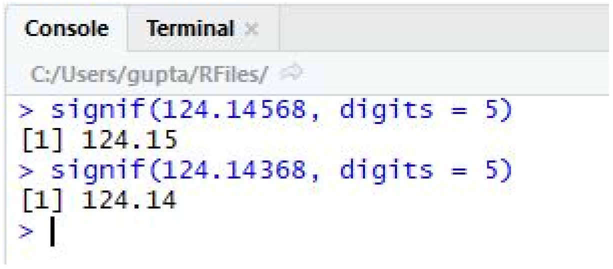
C:/Users/gupta/RFiles/

```
> round(1234.1234876, digits = -3)
[1] 1000
> round(1234.1234876, digits = -2)
[1] 1200
> round(1234.1234876, digits = -1)
[1] 1230
> 
```

Notice that the use of negative numbers in digits only affect the integer portion of the number.

Example No. 1.18 Using the signif() function for rounding

The **signif()** function allows retaining a specific number of digits regardless of the size of the number. Logically, more digits than what the number has cannot be retained. The use of this function is illustrated next where we retain only 5 digits. In all cases, R rounds the number according to the rule mentioned before.



```
> signif(124.14568, digits = 5)
[1] 124.15
> signif(124.14368, digits = 5)
[1] 124.14
>
```

In both cases, 5 digits need to be retained. However, notice that in the first part of the example, the first digit to be dropped is 5, so the previous digit, 4 in this case, is rounded up to 5. In the second part of the example, the first digit to be dropped is 3, so the previous digit, 4 in this case, is left "as is".

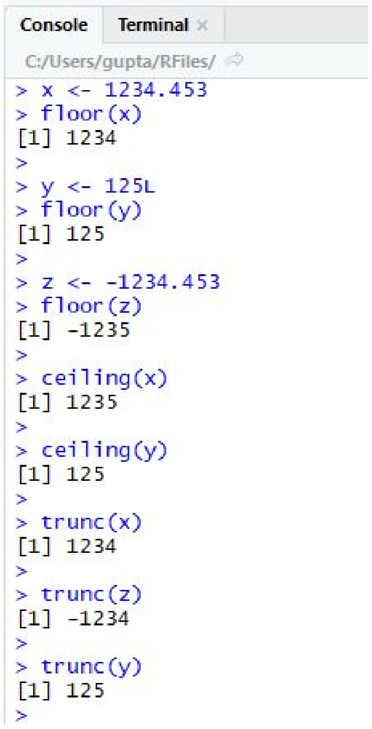
Note No. 1.3 Note that for rounding off a 5, the IEC 60559 standard is expected to be used, "go to the even digit." Therefore, round(0.5) results in 0 and round(-1.5) produces -2. However, this is dependent on the OS services and on its internal error manipulation (since e.g. 0.15 is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so round(0.15, 1) could result, depending on the OS, in 0.1 or 0.2.

1.16.1 The Ceiling(), Floor(), and Trunc() functions

In addition to the round() and signif() functions there are three other functions that can also be used for rounding, however, the realities of computer arithmetic can cause unexpected results, especially with floor and ceiling. Detailed information can be found by using help command **?round()** at the R console. These functions behave as follows:

1. Ceiling() takes a single numeric argument and always returns the smallest integer that is greater or equal to the argument. The ceiling of any integer number is itself.
2. Floor() takes a single numeric argument and always returns the largest integer that is smaller or equal to the argument. The floor() of an integer number is itself.
3. Trunc() takes a single numeric argument and always returns an integer value obtained after truncating the values of its argument "toward zero."

Example No. 1.19 Using the Ceiling(), Floor(), and Trunc() functions



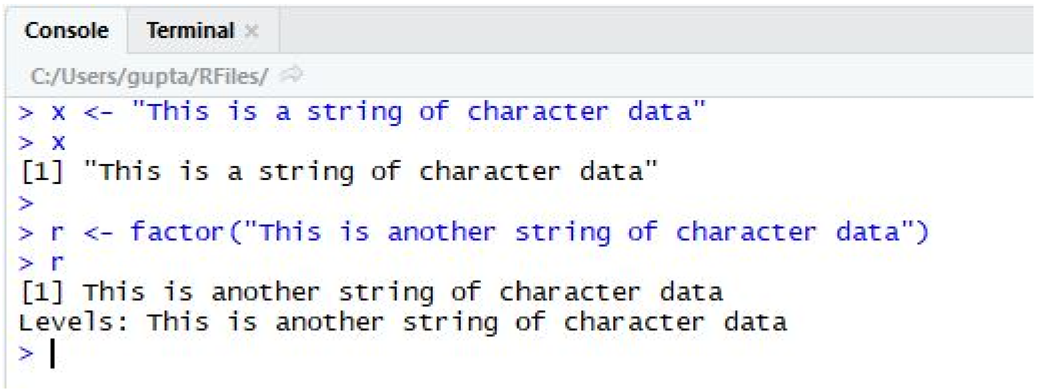
```
> x <- 1234.453
> floor(x)
[1] 1234
>
> y <- 125L
> floor(y)
[1] 125
>
> z <- -1234.453
> floor(z)
[1] -1235
>
> ceiling(x)
[1] 1235
>
> ceiling(y)
[1] 125
>
> trunc(x)
[1] 1234
>
> trunc(z)
[1] -1234
>
> trunc(y)
[1] 125
>
```

The results of these operations, as shown above, do not change the value of the argument, just how the result is presented.

1.17 Character Data

R also offers the convenience of working with characters data. However, R has two primary functions to handle characters and they work very differently. The easiest way to create a character variable is to assign to it a string enclosed in quotes (singles or doubles). The other way of assigning values to a character variables is by means of the R function **factor()** which creates a vector. We will consider vectors in Chapter 2. The function **is.character** can be used to determine if a variable is of type character. The following examples illustrate how to assign values to a character variable and the use of these functions.

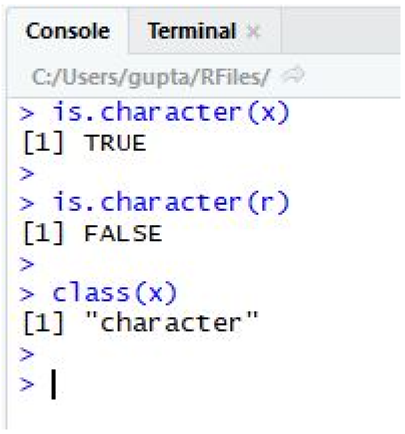
Example No. 1.20 Assigning values to character variables



```
> x <- "This is a string of character data"
> x
[1] "This is a string of character data"
>
> r <- factor("This is another string of character data")
> r
[1] This is another string of character data
Levels: This is another string of character data
> |
```

Observe that the character variables "x" and "r" are both assigned values enclosed in quotes. However, "x" is a character variable whereas "r" is a vector. Notice also that the content of "r" is accompanied by another line of information about the "Levels" of this variable. To confirm this fact even more, the R function **is.character()** can be used on both of these variables to find out more about their nature. The function **class()** can also be used to determine if a variable is of type character. The following example illustrates this.

Example No. 1.21 Using the R function is.character()



```
> is.character(x)
[1] TRUE
>
> is.character(r)
[1] FALSE
>
> class(x)
[1] "character"
>
> |
```

1.18 Summary

This chapter introduced you to the R language and some of the reasons why it is so suitable for data science. We also considered the RStudio IDE and its basic functionality. The chapter also introduced some of the basic arithmetic operations in R as well as the hierarchy of its operators. In addition, variables of the numeric type were also considered and some of the functions that allow us to determine the type and characteristics of these variables. The next chapters will cover some of the basic data structures that make R a very versatile language for data manipulation.