Chapter No. 1
"R Data Types and Basic Operations"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "R Data Types and Basic Operations"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Jaynal Abedin currently holds the position of Statistician at the Centre for Communicable Diseases (CCD) at icddr,b (www.icddrb.org). He attained his Bachelor's and Master's degrees in Statistics from the University of Rajshahi, Rajshahi, Bangladesh. He has vast experience in R programming and Stata and has efficient leadership qualities. He is currently leading a team of statisticians. He has hands-on experience in developing training material and facilitating training in R programming and Stata along with statistical aspects in public health research. His primary area of interest in research includes causal inference and machine learning. He is currently involved in several ongoing public health research projects and is a co-author of several work-in-progress manuscripts. In the useR! Conference 2013, he presented a poster—edeR: Email Data Extraction using R, available at http://www.edii.uclm.es/~useR-2013/abstracts/files/34_edeR_Email_Data_Extraction_using_R.pdf—and obtained the best application poster award. He is also involved in reviewing scientific manuscripts for the Journal of Applied Statistics (JAS) and the Journal of Health Population and Nutrition (JHPN). He is also a successful freelance statistician on online platforms and has an excellent reputation through his high-quality work, especially in R programming. He can be contacted at joystatru@gmail.com, http://bd.linkedin.com/in/jaynal; his Twitter handle is @jaynal83.
This book, Data Manipulation with R, is aimed at giving intermediate to advanced level users of R (who have knowledge about datasets) an opportunity to use state-of-the-art approaches in data manipulation. This book will discuss the types of data that can be handled using R and different types of operations for those data types. Upon reading this book, readers will be able to efficiently manage and check the validity of their datasets with the effective use of R programming, including specialized packages for data management. Readers will come to know about the split-apply-combine strategy, which is the state-of-the-art approach in data management. This book ends with an introduction to how R can be utilized with different database software.

What This Book Covers

Chapter 1, R Data Types and Basic Operations, discusses the different types of data used in R and their basic operations. Before introducing the data types in this chapter, we will highlight what an object in R is and its mode and class. The mode of an object could be either numeric, character, or logical, whereas its class could be vector, factor, list, data frame, matrix, array, or others. This chapter also highlights how to deal with objects in different modes and how to convert from one mode to another and what caution should be taken during conversion. Missing values in R and how to represent missing character and numeric data types are also discussed here. Along with the data types and basic operations, this chapter sheds light on another important aspect, which is almost never mentioned in other text books—the object naming convention in R. We talk about popular object-naming conventions used in R.

Chapter 2, Basic Data Manipulation, introduces some special features that we need to consider during data acquisition. Then, an important aspect of factor manipulation will be discussed, especially when subsetting a factor variable and how to remove unused factor levels. Date processing is also covered using an efficient R package: lubridate. Dealing with the date variable using the lubridate package is much more efficient than any other existing packages that are designed to work with the date variable. Also, string processing will be highlighted and the chapter ends with a description of subscripting and subsetting.
Chapter 3, *Data Manipulation Using plyr*, introduces the state-of-the-art approach called split-apply-combine to manipulate datasets. Data manipulation is an integral part of data cleaning and analysis. For large data, it is always preferable to perform the operations within the subgroup of a dataset to speed up the process. In R, this type of data manipulation can be done with base functionality, but for large data it requires considerable amount of coding and eventually takes more processing time. In the case of large datasets, we can split the data and perform the manipulation or analysis and then again combine them into a single output. This chapter contains a discussion on the different functions in the plyr package that are used for group-wise data manipulation and also for data analysis.

Chapter 4, *Reshaping Datasets*, deals with the orientation of datasets. Reshaping data is a common and tedious task in real-life data manipulation and analysis. A dataset might come with different levels of grouping and we need some reorientation to perform certain types of analysis. To perform statistical analysis, we sometimes require wide data and sometimes long data, and in that case we need to be able to fluently and fluidly reshape data to meet the requirements. Important functions from the reshape package will be discussed in this chapter with examples.

Chapter 5, *R and Databases*, talks about dealing with database software and R. One of the major problems in R is that its memory is bound by RAM, and that is why working with a dataset requires the data to be smaller than its memory. But in reality, the dataset is larger than the capacity of RAM and sometimes the length of arrays or vectors exceeds the maximum addressable range. To overcome these two limitations, R can be utilized with databases. Interacting with databases using R and dealing with large datasets with specialized packages and data manipulation with sqldf will be discussed with examples in this chapter.

*Bibliography*, provides a list of citations used in the book.
R is an object-oriented programming language that is a variation of the S language, and was written by Ross Ihaka and Robert Gentleman (hence the name R), the R Core Development Team, and an army of volunteers. What can we do using R? The answer is we can do anything we can think of that is logical and/or structural. With R, we can perform data processing, write functions, produce graphs, perform complex data analysis, and also produce our own customized packages (a collection of functions for performing specified tasks) to solve specific problems. We can develop up-to-date statistical techniques through R packages, and most importantly, R is open source and a freely available software and it will remain free.

Assuming you have preliminary knowledge on where to get R and how to install it, we will discuss R data types and different operations related to data types. But before introducing data types, we will briefly discuss R objects, modes, and classes because whenever we work in R, we have to deal with these three terminologies frequently. In this chapter, we are going to discuss the following:

- Modes and classes of R objects
- R object structure and mode conversion
- Vector
- Factor and its types
- Data frames, matrices, and arrays
- Lists
- Missing values in R

For More Information:
Modes and classes of R objects

Whatever we do in R, R stores as objects. An R object is anything that can be assigned to a variable of interest. This could be a single number or a set of numbers, characters, and special characters; for example, TRUE, FALSE, NA, NaN, and Inf. Also, these can be the things that are already defined in R as functions, such as seq (to generate a sequence of numbers with a specified increment), names (to extract names such as variable names from a dataset), row.names (to extract the row names of the data, if any), or col.names (this is equivalent to names and it extracts column names from a matrix or data frame). Some of the examples of R objects are as shown in the following code:

```r
# Constant
2
[1] 2
"July"
[1] "July"
NULL
NA
[1] NA
NaN
[1] NaN
Inf
[1] Inf
# Object can be created from existing object
# to make the result reproducible means every time we run the
# following code we will get the same results # we need to set
# a seed value
set.seed(123)
norm(9) + runif(9)
[1] -0.2325549  0.7243262  2.4482476  0.7633118  0.7697945
2.7093348  1.1166220 -0.5565308 -0.1427868
```

One important thing about objects in R is that if we do not assign an object to any variable, we will not be able to re-use it and it does not store the object internally. In the preceding example, all are different objects, but they are not assigned to any variable so they are not stored and we cannot use them later until we enter the object's value itself. So whenever we deal with an object, we will assign it to an appropriate variable, and interestingly the assigned variable is also an object in R!
To assign an object in R to a variable, we can define the variable name in various ways, such as lowercase, uppercase, a combination of upper and lowercase, or even a combination of uppercase, lowercase, and a number and/or a dot; but there are some rules to define variable names. For example, the name cannot start with numbers; it will start with a character or underscore. There is no special character allowed in variable names, such as @, #, $, and *. Though R does not have a standard guideline for naming conventions, according to Bååth (in the paper *The State of Naming Conventions in R*, which can be found at http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf), the most popular function naming convention is lower CamelCase while the most popular naming convention for arguments is period separated. For a variable name, we can use the same naming convention as that of arguments, but again there is no strict rule for naming conventions in R. The following table is reconstructed from the same article by Bååth to give you an idea of the different naming conventions used in R and their popularity:

<table>
<thead>
<tr>
<th>Object type</th>
<th>Naming conventions</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>lowerCamelCase</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>period.separated</td>
<td>51.8</td>
</tr>
<tr>
<td></td>
<td>underscore_separated</td>
<td>37.4</td>
</tr>
<tr>
<td></td>
<td>singlelowercaseword</td>
<td>32.2</td>
</tr>
<tr>
<td></td>
<td>_OTHER.conventions</td>
<td>12.8</td>
</tr>
<tr>
<td></td>
<td>UpperCamelCase</td>
<td>6.9</td>
</tr>
<tr>
<td>Parameter (argument)</td>
<td>period.separated</td>
<td>82.8</td>
</tr>
<tr>
<td></td>
<td>lowerCamelCase</td>
<td>75.0</td>
</tr>
<tr>
<td></td>
<td>underscore_separated</td>
<td>70.7</td>
</tr>
<tr>
<td></td>
<td>singlelowercaseword</td>
<td>69.6</td>
</tr>
<tr>
<td></td>
<td>_OTHER.conventions</td>
<td>9.7</td>
</tr>
<tr>
<td></td>
<td>UpperCamelCase</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Once we store the R object into a variable, it is still treated as an R object. Each and every object in R has some attributes to describe the nature of the information contained in it. The mode and class are the most important attributes of an R object. Commonly encountered modes of an individual R object are numeric, character, and logical. When we work with data in R, problems might arise due to incorrect operations in incorrect object modes. So before working with data, we should study the mode; we need to know what type of operation is applicable.

The `mode` function returns the mode of R objects. The following example code describes how we can investigate the mode of an R object:

```r
# Storing R object into a variable and then viewing the mode
num.obj <- seq(from=1,to=10,by=2)
mode(num.obj)
[1] "numeric"

logical.obj <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
mode(logical.obj)
[1] "logical"

character.obj <- c("a","b","c")
mode(character.obj)
[1] "character"
```

For the numeric mode, R stores all numeric objects into either a 32-bit integer or double-precision floating point.

If an R object contains both numeric and logical elements, the mode of that object will be numeric and in that case the logical element automatically gets converted to numeric. The logical element `TRUE` converts to 1 and `FALSE` converts to 0. On the other hand, if any R object contains a character element along with both numeric and logical elements, it automatically converts to the character mode. Let's have a look at the following code:

```r
# R object containing both numeric and logical element
xz <- c(1, 3, TRUE, 5, FALSE, 9)
xz
[1] 1 3 1 5 0 9
mode(xz)
[1] "numeric"

# R object containing character, numeric, and logical elements
xw <- c(1,2,TRUE,FALSE,"a")
xw
[1] "1" "2" "TRUE" "FALSE" "a"
mode(xw)
[1] "character"
```
The `mode()` function is not the only way to test R object modes; there are alternative ways too, which are `is.numeric()`, `is.character()`, and `is.logical()`, as shown in the following code. The output of these functions is always logical.

```r
num.obj <- seq(from=1, to=10, by=2)
logical.obj <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
character.obj <- c("a", "b", "c")

is.numeric(num.obj)
[1] TRUE
is.logical(num.obj)
[1] FALSE
is.character(num.obj)
[1] FALSE
```

Other than these three modes (numeric, logical, and character) of objects, another frequently encountered mode is function; for example:

```r
c mode(mean)
[1] "function"
# Also we can test whether "mean" is function or not as follows
is.function(mean)
[1] TRUE
```

The `class()` function provides the class information of an R object. The primary purpose of the `class()` function is to know how different functions, including generic functions, will work. For example, with the class information, the generic function `print` or `plot` knows what to do with a particular R object. To assess the class information of the object created earlier, we can use the `class()` function. Let's have a look at the following code:

```r
num.obj <- seq(from=1, to=10, by=2)
logical.obj <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
character.obj <- c("a", "b", "c")

class(num.obj)
[1] "numeric"

class(logical.obj)
[1] "logical"

class(character.obj)
[1] "character"
```
Although we can easily assess the mode and class of an R object through `mode()` and `class()`, there is another collection of R commands that are also used to assess whether a particular object belongs to a certain class. These functions start with `is`, for example; `is.numeric()`, `is.logical()`, `is.character()`, `is.list()`, `is.factor()`, and `is.data.frame()`. As R is an object-oriented programming language, there are many functions (collectively known as generic functions) that will behave differently depending on the class of that particular object.

The mode of an object tells us how it's stored. It could happen that two different objects are stored in the same mode with different classes. How those two objects are printed using the `print` command is determined by its class; for example:

```r
# Output omitted due to space limitation
num.obj <- seq(from=1,to=10,by=2)
set.seed(1234) # To make the matrix reproducible
mat.obj <- matrix(runif(9),ncol=3,nrow=3)
mode(num.obj)
mode(mat.obj)
class(num.obj)
class(mat.obj)
# prints a numeric object
print(num.obj)
# prints a matrix object
print(mat.obj)
```

Like `character` and `numeric`, there is another method you can use to store data when the data is categorical in nature. In categorical data, we usually have some unique values and their corresponding labels. To store this type of object in R, we use the class `factor`, which allows less storage location because it is required to store only unique levels once.

Interestingly, once we try to see the mode of a `factor` object, it always shows `numeric` even if it displays character data. For example:

```r
character.obj <- c("a","b","c")
character.obj
[1] "a"  "b"  "c"

is.factor(character.obj)
[1] FALSE

# Converting character object into factor object using as.factor()
factor.obj <- as.factor(character.obj)
```

---

For More Information:
factor.obj  
[1] a b c  
Levels: a b c

is.factor(factor.obj)  
[1] TRUE

mode(factor.obj)  
[1] "numeric"

class(factor.obj)  
[1] "factor"

We have to be careful when dealing with the factor class data in R. The important thing to remember is that for vectors (we will discuss vectors in the Vector section in this chapter), the class and mode will always be numeric, logical, or character. On the other hand, for matrices and arrays (we will discuss matrices and arrays in the Factor and its type section in this chapter), a class is always a matrix or array, but its mode can be numeric, character, or logical.

**R object structure and mode conversion**

When we work with any statistical software, such as R, we rarely use single values for an object. We need to know how we can handle a collection of data values (for example, the age of 100 randomly selected diabetic patients) along with what type of objects need to store those data values. In R, the most convenient way to store more than one data value is vector (a collection of data values stored in a single object is known as a vector; for example, storing the ages of 100 diabetic patients in a single object). In fact, whenever we create an R object, it stores the values as a vector. It could be a single-element vector or multiple-element vector. The num.obj vector we have created in the previous section is a kind of vector comprising of numeric elements.

One of the simplest ways to create a vector in R is to use the c() function. For example:

```r
# creating vector of numeric element with "c" function
num.vec <- c(1,3,5,7)
num.vec
[1] 1 3 5 7
mode(num.vec)
[1] "numeric"
class(num.vec)
[1] "numeric"
is.vector(num.vec)
[1] TRUE
```

For More Information:

If we create a vector with mixed elements (character and numeric), the resulting vector will be a character vector. For example:

```r
# Vector with mixed elements
num_char_vec <- c(1,3,"five",7)
num_char_vec
[1] "1" "3" "five" "7"
mode(num_char_vec)
[1] "character"
class(num_char_vec)
[1] "character"
is.vector(num_char_vec)
[1] TRUE
```

We can create a big new vector by combining multiple vectors, and the resulting vector's mode will be character if any element of any vector contains a character. The vector could be named or without a name; in the previous example, vectors were without names. The following example shows how we can create a vector with the name of each element:

```r
# combining multiple vectors
comb_vec <- c(num_vec,num_char_vec)
mode(comb_vec)
[1] "character"

# creating named vector
named_num_vec <- c(x1=1,x2=3,x3=5)
named_num_vec
x1 x2 x3
 1 3  5
```

The name of the elements in a vector can be assigned separately using the `names()` command. In R, any single constant is also stored as a vector of the single element. For example:

```r
# vector of single element
unit_vec <- 9
is.vector(unit_vec)
[1] TRUE
```
R has six basic storage types of vectors and each type is known as an atomic vector. The following table shows the six basic vector types, their mode, and the storage mode:

<table>
<thead>
<tr>
<th>Type</th>
<th>Mode</th>
<th>Storage mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>logical</td>
<td>logical</td>
</tr>
<tr>
<td>integer</td>
<td>numeric</td>
<td>integer</td>
</tr>
<tr>
<td>double</td>
<td>numeric</td>
<td>double</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>character</td>
<td>character</td>
<td>character</td>
</tr>
<tr>
<td>raw</td>
<td>raw</td>
<td>raw</td>
</tr>
</tbody>
</table>

Other than vectors, there are different storage types available in R to handle data with multiple elements, which are matrix, dataframe, and list. We will discuss each of these types in the subsequent sections.

To convert the object mode, R has user friendly functions that can be depicted as follows: as.x. Here, x could be numeric, logical, character, list, data.frame, and so on. For example, if we need to perform a matrix operation that requires the numeric mode and the data is stored in some other mode, the operation cannot be performed. In that case, we need to convert that data into the numeric mode.

In the following example, we will see how we can convert an object's mode:

```r
# creating a vector of numbers and then converting it to logical # and character
numbers.vec <- c(-3,-2,-1,0,1,2,3)
nnumbers.vec
[1] -3 -2 -1  0  1  2  3
num2char <- as.character(numbers.vec)
num2char
[1] "-3" "-2" "-1" "0" "1" "2" "3"
num2logical <- as.logical(numbers.vec)
num2logical
[1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE

# creating a character vector and then convert it to numeric and logical
```

For More Information:
R Data Types and Basic Operations

```
char.vec <- c("1","3","five","7")
char.vec
[1] "1"  "3"  "five" "7"
char2num <- as.numeric(char.vec)
Warning message:
NAs introduced by coercion
char2num
[1]  1  3 NA  7
char2logical <- as.logical(char.vec)
char2logical
[1] NA NA NA NA

# logical to character conversion
logical.vec <- c(TRUE, FALSE, FALSE, TRUE, TRUE)
logical.vec
[1] TRUE FALSE FALSE TRUE TRUE
logical2char <- as.character(logical.vec)
logical2char
[1] "TRUE" "FALSE" "FALSE" "TRUE" "TRUE"
```

Note that when we convert the numeric mode to the logical mode, only 0 (zero) gets FALSE and all the other values get TRUE. Also, if we convert a character object to numeric, it produces numeric elements and NA (if any actual character is present), and a warning will be issued. Importantly, R does not convert a character object into a logical object, but if we try to do this, all the resulting elements will be NA. However, logical objects get successfully converted to character objects. Finally, we can say that any object can be converted to a character without any warning, but if we want to convert character objects to any other type, we have to be careful.

Vector

The R vector can be contiguous cells containing data. In R, the basic data storage type is vector. The vector itself could be numeric, character, and logical based on the elements. In fact, there are six types of vectors used in R. We can easily access elements of a vector through indexing. The following example shows how we can create a vector and access its individual elements and group of elements:

```
# creating a vector and accessing elements
vector1 <- c(1,3,5,7,9)
vector1
[1] 1 3 5 7 9
```

For More Information:
# accessing second elements of "vector1"
vector1[2]
[1] 3

# accessing three elements starting from second element
vector1[2:4]
[1] 3 5 7

# another way of creating vector. Here "from" is the starting point # of the vector and "to" is the end point of the vector and "by" is # increment
vector2 <- seq(from=2, to=10, by=2)
is.vector(vector2)
[1] TRUE

**Factor and its types**

A factor is another important data type in R, especially when we deal with categorical variables. In an R vector, there is no limit on the number of distinct elements, but in factor variables, it takes only a limited number of distinct elements. This type of variable is usually referred to as a categorical variable during data analysis and statistical modeling. In statistical modeling, the behavior of a numeric variable and categorical variable is different, so it is important to store the data correctly to ensure valid statistical analysis.

In R, a factor variable stores distinct numeric values internally and uses another character set to display the contents of that variable. In other software, such as Stata, the internal numeric values are known as values and the character set is known as value labels. Previously, we saw that the mode of a factor variable is numeric; this is due to the internal values of the factor variable.

A factor variable can be created using the `factor` command; the only required input is a vector of values, which will return as a vector of factor values. The input can be numeric or character, but the levels of factor will always be a character. The following example shows how to create factor variables:

```r
#creating factor variable with only one argument with factor()
factor1 <- factor(c(1,2,3,4,5,6,7,8,9))
factor1
[1] 1 2 3 4 5 6 7 8 9
Levels: 1 2 3 4 5 6 7 8 9
levels(factor1)
```

For More Information:
R Data Types and Basic Operations

```r
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
labels(factor)
[1] "1"
labels(factor1)
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"

#creating factor with user given levels to display
factor2 <- factor(c(1,2,3,4,5,6,7,8,9),labels=letters[1:9])
factor2
[1] a b c d e f g h i
Levels: a b c d e f g h i
levels(factor2)
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
labels(factor2)
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

In a factor variable, the values themselves are stored as numeric vectors, whereas the labels store only unique characters, and it stores only once for each unique character. Factors can be ordered if the `ordered=T` command is specified, otherwise it inherits the order of the levels specified.

A factor could be numeric with numeric levels, but direct mathematical operations are not possible with this numeric factor. Special care should be taken if we want to use mathematical operations. The following example shows a numeric factor and its mathematical operation:

```r
# creating numeric factor and trying to find out mean
num.factor <- factor(c(5,7,9,5,6,7,3,5,3,9,7))
num.factor
[1] 5 7 9 5 6 7 3 5 3 9 7
Levels: 3 5 6 7 9
mean(num.factor)
[1] NA
Warning message:
In mean.default(num.factor) :
  argument is not numeric or logical: returning NA
```

From the preceding example, we see that we can create a numeric factor, but the mathematical operation is not possible. And when we tried to perform a mathematical operation, it showed us a warning and produced the result `NA`. To perform any mathematical operation, we need to convert the factor to its numeric counterpart. One can assume that we can easily convert the factor to numeric using the `as.numeric()` function, but if we use the `as.numeric()` function, it will only convert the internal values of the factors, not the desired values.

For More Information:
So the conversion must be done with levels of that factor variable; optionally, we can firstly convert the factor into a character using `as.character()` and then use `as.numeric()`. The following example describes the scenario:

```r
num.factor <- factor(c(5, 7, 9, 5, 6, 7, 3, 5, 3, 9, 7))
num.factor
[1] 5 7 9 5 6 7 3 5 3 9 7
Levels: 3 5 6 7 9
#as.numeric() function only returns internal values of the factor
as.numeric(num.factor)
[1] 2 4 5 2 3 4 1 2 1 5 4
# now see the levels of the factor
levels(num.factor)
[1] "3" "5" "6" "7" "9"
as.character(num.factor)
[1] "5" "7" "9" "5" "6" "7" "3" "5" "3" "9" "7"

# now to convert the "num.factor" to numeric there are two method
# method-1:
mean(as.numeric(as.character(num.factor)))
[1] 6

# method-2:
mean(as.numeric(levels(num.factor)[num.factor]))
[1] 6
```

**Data frame**

A data frame is a rectangular arrangement of rows and columns of vectors and/or factors, such as a spreadsheet in MS Excel. The columns represent variables in the data and the rows represent observations or records. In other software, such as a database package, each column represents a field and each row represents a record.

Dealing with data does not mean dealing with only one vector or factor variable, rather it is the collection of variables. Each column represents only one type of data: numeric, character, or logical, and each row represents case information across all columns. One important thing to remember about R data frames is that all vectors should be of the same length. In an R data frame, we can store different types of variables, such as numeric, logical, factor, and character. To create a data frame, we can use the `data.frame()` command. The following example shows how to create a data frame using different vectors and factors:

```r
#creating vector of different variables and then creating data frame
var1 <- c(101, 102, 103, 104, 105)
```
R Data Types and Basic Operations

```r
var2 <- c(25, 22, 29, 34, 33)
var3 <- c("Non-Diabetic", "Diabetic", "Non-Diabetic", "Non-Diabetic", "Diabetic")
var4 <- factor(c("male", "male", "female", "female", "male"))
# now we will create data frame using two numeric vectors one
# character vector and one factor
diab.dat <- data.frame(var1, var2, var3, var4)
diab.dat

var1 var2         var3   var4
1  101   25 Non-Diabetic   male
2  102   22     Diabetic   male
3  103   29 Non-Diabetic female
4  104   34 Non-Diabetic female
5  105   33     Diabetic   male
```

Now if we see the class of individual columns of the newly created data frame, we will see that the first two columns' classes are numeric and the last two columns' classes are factor, though initially the class of `var3` was character. One thing is obvious here—when we create data frames and any one of the column's classes is character, it automatically gets converted to factor, which is a default R operation. But there is one argument, `stringsAsFactors=FALSE`, that allows us to prevent the automatic conversion of character to factor during data frame creation. In the following example, we will see this:

```r
# class of each column before creating data frame
class(var1)
[1] "numeric"
class(var2)
[1] "numeric"
class(var3)
[1] "character"
class(var4)
[1] "factor"
```

To access individual columns (variables) from a data frame, we can use a dollar ($) sign along with the data frame name; for example, `diab.dat$var1`. There are some other ways to access variables from a data frame, such as the following:

- Data frame name followed by double square brackets with variable names within quotation marks; for example, `diab.dat["var1"]`
- Data frame name followed by single square brackets with the column index; for example, `diab.dat[,1]`

For More Information:

Besides these, there is one other way that allows us to access each of the individual variables as separate objects. The R `attach()` function allows us to access individual variables as separate R objects. Once we use the `attach()` command, we need to use `detach()` to remove individual variables from the working environment. Let's have a look at the following code:

```r
# class of each column after creating data frame
class(diab.dat$var1)
[1] "numeric"
class(diab.dat$var2)
[1] "numeric"
class(diab.dat$var3)
[1] "factor"
class(diab.dat$var4)
[1] "factor"
# now create the data frame specifying as.is=TRUE
diab.dat.2 <- data.frame(var1, var2, var3, var4, stringsAsFactors=FALSE)
diab.dat.2
   var1 var2         var3   var4
1  101  25 Non-Diabetic   male
2  102  22     Diabetic   male
3  103  29 Non-Diabetic female
4  104  34 Non-Diabetic female
5  105  33     Diabetic   male
class(diab.dat.2$var3)
[1] "character"
```

**Matrices**

A matrix is also a two-dimensional arrangement of data but it can take only one class. To perform any mathematical operations, all columns of a matrix should be numeric. However, in data frames we can store numeric, character, or factor columns. To perform any mathematical operation, especially a matrix operation, we can use matrix objects. However, in data frames, we are unable to perform certain types of mathematical operations, such as matrix multiplication. To create a matrix, we can use the `matrix()` command or convert a numeric data frame to a matrix using `as.matrix()`. We can convert the data frame that we created earlier as `diab.dat` to a matrix using `as.matrix()`, but this is not suitable to perform mathematical operations, as shown in the following example:

```r
# data frame to matrix conversion
mat.diab <- as.matrix(diab.dat)
```
R Data Types and Basic Operations

```r
mat.diab
  var1 var2 var3           var4
[1,] "101" "25" "Non-Diabetic" "male"
[2,] "102" "22" "Diabetic"    "male"
[3,] "103" "29" "Non-Diabetic" "female"
[4,] "104" "34" "Non-Diabetic" "female"
[5,] "105" "33" "Diabetic"    "male"

class(mat.diab)
[1] "matrix"
mode(mat.diab)
[1] "character"

# matrix multiplication is not possible with this newly created matrix

t(mat.diab) %*% mat.diab
Error in t(mat.diab) %*% mat.diab :
  requires numeric/complex matrix/vector arguments

# creating a matrix with numeric elements only
# To produce the same matrix over time we set a seed value
set.seed(12345)
num.mat <- matrix(rnorm(9), nrow=3, ncol=3)
num.mat
  [,1]       [,2]       [,3]
[1,] 0.5855288 -0.4534972  0.6300986
[2,] 0.7094660  0.6058875 -0.2761841
[3,] -0.1093033 -1.8179560 -0.2841597
class(num.mat)
[1] "matrix"
mode(num.mat)
[1] "numeric"

# matrix multiplication
t(num.mat) %*% num.mat
  [,1]       [,2]       [,3]
[1,] 0.8581332 0.36302951 0.20405722
[2,] 0.3630295 3.87772320 0.06350551
[3,] 0.2040572 0.06350551 0.55404860
```

For More Information:
Arrays

An array is a multiply-subscripted data entry that allows the storing of data frames, matrices, or vectors of different types. Data frames and matrices are of two dimensions only, but an array could be of any number of dimensions. Sometimes, we need to store multiple matrices or data frames into a single object; in this case, we can use arrays to store this data. The following is a simple example to store three matrices of order 2x2 in a single array object:

```r
mat.array=array(dim=c(2,2,3))

# To produce the same results over time we set a seed value
set.seed(12345)

mat.array[,,1]<-rnorm(4)
mat.array[,,2]<-rnorm(4)
mat.array[,,3]<-rnorm(4)

mat.array

[, 1]
[1,]  0.5855288 -0.1093033
[2,]  0.7094660 -0.4534972

[, 2]
[1,]  0.6058875  0.6300986
[2,] -1.8179560 -0.2761841

[, 3]
[1,] -0.2841597 -0.1162478
[2,] -0.9193220  1.8173120
```

For More Information:
**list**

A list object is a generic R object that can store other objects of any type. In a list object, we can store single constants, vectors of numeric values, factors, data frames, matrices, and even arrays. Recalling the vectors var1, var2, var3, and var4; the data frame created using these vectors; and also recalling the array created in the Arrays section, we will create a list object in the following example:

```r
var1 <- c(101,102,103,104,105)
var2 <- c(25,22,29,34,33)
var3 <- c("Non-Diabetic","Diabetic","Non-Diabetic","Non-Diabetic","Diabetic")
var4 <- factor(c("male","male","female","female","male"))
diab.dat <- data.frame(var1,var2,var3,var4)

mat.array=array(dim=c(2,2,3))
set.seed(12345)
mat.array[,1]<-rnorm(4)
mat.array[,2]<-rnorm(4)
mat.array[,3]<-rnorm(4)

# creating list
obj.list <- list(elem1=var1,elem2=var2,elem3=var3,elem4=var4,elem5=diab.dat,elem6=mat.array)

obj.list

$obj1
[1] 101 102 103 104 105

$obj2
[1] 25 22 29 34 33

$obj3
[1] "Non-Diabetic" "Diabetic" "Non-Diabetic" "Non-Diabetic" "Diabetic"

$obj4
[1] male female female male
Levels: female male
```
To access individual elements from a list object, we could use the name of that component or use double square brackets with the index of those elements. For example, `obj.list[[1]]` will give the first element of the newly created list object.

**Missing values in R**

Missing values are part of the data manipulation process and we will encounter some missing values in almost every dataset. So, it is important to know how R handles missing values and how they are represented. In R, a numeric missing value is represented by `NA` while character missing values are represented by `<NA>`. To test if there is any missing value present in a dataset (data frame), we can use `is.na()` for each column or we can use this function in combination with the `any()` function. The following example shows how we can see if there are any missing values present in a dataset:

```r
missing_dat <- data.frame(v1=c(1,NA,0,1),v2=c("M","F",<NA>,"M"))
missing_dat
```

For More Information:

R Data Types and Basic Operations

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
</tr>
</tbody>
</table>

is.na(missing_dat$v1)
[1] FALSE TRUE FALSE FALSE

is.na(missing_dat$v2)
[1] FALSE FALSE TRUE FALSE

any(is.na(missing_dat))
[1] TRUE

Summary
In this chapter, we firstly talked very briefly about what R is. We did not cover where to get it and how to install it as we are assuming the reader will have some preliminary knowledge in those areas. Then we introduced what R objects are and their modes and classes. We also highlighted how we can convert modes of objects using R functions such as `as.numeric` and `as.character`. Finally, we discussed different R objects, such as vector, factor, data frame, matrix, and list. The chapter ended with an introduction to how missing values are represented and dealt with in R. In the next chapter, we will discuss data manipulation with different R objects in greater detail.
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.