
# Data Manipulation with R

Perform group-wise data manipulation and deal with large datasets using R efficiently and effectively

Jaynal Abedin

[PACKT] open source
PUBLISHING

# Table of Contents

# Data Manipulation with R

# Data Manipulation with R



First published: January 2014

Production Reference: 1080114

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-109-1

www.packtpub.com

Cover Image by Huzzatul Mursalin (<apu_mb@yahoo.com>)

# Credits

**Author**

Jaynal Abedin

**Reviewers**

A. Dhandapani

Colman McMahon

Vignesh Prajapati

**Acquisition Editors**

Kartikey Pandey

Owen Roberts

**Commissioning Editor**

Priyanka Shah

**Technical Editors**

Manan Badani

Ankita Jha

**Copy Editor**

Aditya Nair

**Project Coordinator**

Sageer Parkar

**Proofreader**

Maria Gould

**Indexer**

Rekha Nair

**Production Coordinator**

Nitesh Thakur

**Cover Work**

Nitesh Thakur

# 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.

# About the Reviewers

**A. Dhandapani** is currently working as a professor of Statistics and Computer Applications at the National Academy of Agricultural Research Management (NAARM), Hyderabad, India. He holds a Master's and a Ph.D. degree in Agricultural Statistics from the Indian Agricultural Research Institute, New Delhi, specializing in sampling techniques. He joined the Agricultural Research Service of the Indian Council of Agricultural Research in the discipline of Agricultural Statistics in 1996 and was posted as a scientist at New Delhi. He has worked in the area of pest surveillance, pest forewarning models, and developed several information systems (both web-and desktop-based) in the area of plant protection. He has also developed a web-based application for the generation of Hadamard Matrices. Currently, he is teaching business management students of Business Statistics, Business Analytics, Marketing Research, Management Information System, and Enterprise Resource Planning for Agribusiness. Besides this, he trains agricultural scientists in data analysis using SAS. He is also involved in creating information systems for the collection and analysis of data collected from large-scale coordinated trials in agriculture.

**Colman McMahon** is a PhD fellow in the Dynamics Lab, University College, Dublin. His research is on policy network analysis and data visualization of the EU's proposed General Data Protection Regulation. This project is a part of the Simulation Science program run under the auspices of the Complex Adaptive Systems Laboratory (CASL). In addition to research, he has lectured on data visualization and knowledge management for M.Sc Computer Science courses. Prior to his full immersion into academia, he has worked in visual effects for film and television in Los Angeles and was an independent technology consultant.

**Vignesh Prajapati** is a Big Data scientist at Pingax. He loves to play with open source technologies like R, Hadoop, MongoDB, and Java. He has been working on data analytics with machine learning, R, Hadoop, RHadoop, and MongoDB. He has expertise in algorithm development for Data ETL and generating recommendations, prediction, and behavioral targeting over e-commerce historical Google Analytics, and other datasets. He has also written several articles on R, Hadoop, and machine learning fields for producing intelligent Big Data applications. He can be contacted at <vignesh2066@gmail.com> and http://in.linkedin.com/in/vigneshprajapati/.

Vignesh has worked on another book with Packt Publishing. He has written *Big Data Analytics with R and Hadoop* (https://www.packtpub.com/big-data-analytics-with-r-and-hadoop/book).

Firstly, I would like to thank my teachers, who introduced me to this wonderful open source technology during my undergraduate years. Then I would like to express my gratitude to my family, friends, colleagues, and well wishers, who always motivated me to contribute to this technology. Last but not least, I would like to express my deepest gratitude to Packt Publishing and its team, who gave me the opportunity to write this book. Finally, I am grateful to all of the reviewers for their time and constructive suggestions.

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

*Dedicated to my late grandmother*

*--Jaynal Abedin*

# Preface

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.

# What you need for this book

Readers are expected to have basic knowledge of R and some knowledge of statistical data. To run the examples from this book, R should be installed, and it can be found at http://www.r-project.org. The example files are produced on R 2.15.2 and R 3.0.1.

# Who this book is for

This book is for intermediate to advanced level users of R who have knowledge about datasets. Also, this book is for those who regularly deal with different research data, including but not limited to public health, business analysis, and the machine-learning community.

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Once we have an R object we can easily assess its mode by using `mode()`."

A block of code is set as follows:

```
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
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# Calling xlsx library
library(xlsx)
# importing xlsxanscombe.xlsx
anscombe_xlsx <- read.xlsx2("xlsxanscombe.xlsx",sheetIndex=1)
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Add...** button and select an appropriate ODBC driver and then locate the desired file and give a data source name."

## Note

Warnings or important notes appear in a box like this.

## Tip

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

# Chapter 1. R Data Types and Basic Operations

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

## 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:

```
# Constant
2
[1] 2
"July"
[1] "July"
NULL
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)
rnorm(9)+runif(9)
[1] -0.2325549  0.7243262  2.4482476  0.7633118  0.7697945  2.7093348  1.1166220 -
0.5565308 -0.1427868
```

**Tip**

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

One important thing about objects in R is that if we do not assign an object to any variable, we will not 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:

| **Object type** | **Naming conventions** | **Percentage** |
|---|---|---|
| **Function** | | |
| | lowerCamelCase | 55.5 |
| | period.separated | 51.8 |
| | underscore_separated | 37.4 |
| | singlelowercaseword | 32.2 |
| | _OTHER.conventions | 12.8 |
| | UpperCamelCase | 6.9 |
| **Parameter (argument)** | | |
| | period.separated | 82.8 |

| | | |
|---|---|---|
| | lowerCamelCase | 75.0 |
| | underscore_separated | 70.7 |
| | singlelowercaseword | 69.6 |
| | _OTHER.conventions | 9.7 |
| | UpperCamelCase | 2.4 |

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:

```
# 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 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.charater()`, and `is.logical()`, as shown in the following code. The output of these functions is always logical.

```
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:

```
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:

```
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:

```
# 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:

```
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)
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:

```
# 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
```

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

```
# 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:

```
# 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:

```
# 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:

| **Type** | **Mode** | **Storage mode** |
|---|---|---|
| logical | logical | logical |
| integer | numeric | integer |
| double | numeric | double |
| complex | complex | complex |
| character | character | character |
| raw | raw | raw |

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:

```
# creating a vector of numbers and then converting it to logical # and character
numbers.vec <- c(-3,-2,-1,0,1,2,3)
numbers.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 character vector and then convert it to numeric and logical
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
# 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:

```
#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)
[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:

```
# 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.

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:

```
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:

```
#creating vector of different variables and then creating data frame
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"))
# 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:

```
#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]`

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:

```
# 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:

```
# data frame to matrix conversion
mat.diab <- as.matrix(diab.dat)
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
```

# 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:

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

, , 2

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

, , 3

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

# 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:

```
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
$elem1
[1] 101 102 103 104 105

$elem2
[1] 25 22 29 34 33

$elem3
[1] "Non-Diabetic" "Diabetic"     "Non-Diabetic" "Non-Diabetic" "Diabetic"

$elem4
[1] male   male   female female male
Levels: female male

$elem5
  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

$elem6
, , 1

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

, , 2

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

, , 3

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

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:

```
missing_dat <- data.frame(v1=c(1,NA,0,1),v2=c("M","F",NA,"M"))
missing_dat
  v1   v2
1  1    M
2 NA    F
3  0 <NA>
4  1    M

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.

# Chapter 2. Basic Data Manipulation

When preparing a dataset for statistical analysis, data processing and manipulations such as checking, cleaning, and creating new variables are two of the important tasks. In this chapter, the basics of data manipulation will be discussed with examples that will give us an idea about checking a dataset and cleaning it if necessary. This chapter will deal with the following topics:

- Importing datasets from different sources
- Factor manipulations
- Numeric variables and date processing
- Character manipulations
- Subscripting and subsetting datasets

## Acquiring data

A dataset can be stored in a computer or any other storage device in different file formats. R provides the useful facility to access different file formats through different commands. Some of the commonly used file formats are:

- `Comma separated values (*.csv)`
- `Text file with Tab delimited`
- `MS Excel file (*.xls or *.xlsx)`
- `R data object (*.RData)`

Other than the file formats mentioned in the preceding list, the dataset can be stored in another statistical software format; for example, Stata, SPSS, or SAS. In R, using the `foreign` library, we can acquire a dataset from other statistical software. In the following examples, we will see how we can acquire data in R from different file formats.

Firstly, we will import a `.csv` file, `CSVanscombe.csv`. This file contains four pairs of numeric variables, `(x1,y1)` to `(x4,y4)`. The noticeable feature of this file is that the actual data starts from the third row and the first two rows contain a brief description about the dataset. Now we will use `read.csv()` to import the file and store it in the `anscombe` object in R, which will be a data frame, as shown in the following code:

```
# Before running the following command we need to set the data
# location using setwd(). For example setwd("d:/chap2").

anscombe <- read.csv("CSVanscombe.csv",skip=2)
```

Note that in the preceding code, the `skip=2` command is used, which tells R that the actual data starts from the third row.

If a `.csv` file contains both numeric and character variables and we use `read.csv()`, the character variables get automatically converted to the factor type. We can prevent character variables from this automatic conversion to factor by specifying `stringsAsFactors=FALSE` within the `read.csv()` command, as shown in the following code:

```
# import csv file that contains both numeric and character variable
# firstly using default and then using stringsAsFActors=FALSE

iris_a <- read.csv("iris.csv")
str(iris_a)
'data.frame':   150 obs. of  5 variables:
$ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
$ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

In the following example, we will see the difference if we specify the `stringsAsFactors = FALSE` argument:

```
# Now using stringsAsFactors=FALSE
iris_b <- read.csv("iris.csv",stringsAsFactors=F)
'data.frame':   150 obs. of  5 variables:
$ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
$ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
$ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

We see that in the first data frame, the class of the `Species` variable is `factor`, whereas in the second data frame the class of the same variable is `character`. So we have to be careful while importing the `.csv` file with mixed variables.

Sometimes, it could happen that the file extension is `*.csv` but the data is not comma separated; rather, the data supplier has used a semicolon (`;`) as a separator, or any other symbol. In that case, we can still use the `read.csv()` command, but this time we have to specify the separator. Let's look at the example with a semicolon-separated `.csv` file of the same `iris` data:

```
iris_semicolon <- read.csv("iris_semicolon.csv",stringsAsFactors=FALSE,sep=";")
str(iris_semicolon)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

Similarly, if the values are tab separated, we can use `read.csv()` with `sep= "\t"`. Alternatively, we can use `read.table()`. The following is an example:

```
anscombe_tab <- read.csv("anscombe.txt",sep="\t")
anscombe_tab_2 <- read.table("anscombe.txt",header=TRUE)
```

Notice that here when we used `read.table()`, we have to specify whether the variable name is present or not using `header=TRUE`.

If the dataset is stored in the `*.xls` or `*.xlsx` format, we have to use certain R packages to import those files; one of the packages is `xlsx`, which is designed to read files formatted as `*.xlsx`. The following is an example to import the `xlsxanscombe.xlsx` file:

```
# Calling xlsx library
library(xlsx)
# importing xlsxanscombe.xlsx
anscombe_xlsx <- read.xlsx2("xlsxanscombe.xlsx",sheetIndex=1)
```

In R, single or multiple data frames or other objects can be stored in the `*.RData` format. This file format is convenient to store more than one dataset into a single file. To acquire a dataset for any other type of objects from the `*.RData` file, we can use the `load()` function. The following is an example to load multiple datasets and a vector of R objects from a single `*.RData` file:

```
# loading robjects.RData file
load("robjects.RData")
# to see whether the objects are imported correctly
objects()
"character.obj" "diab.dat" "logical.obj" "num.obj" "var1" "var2" "var3" "var4"
```

Note that the `objects()` command is used to look at all of the objects in the current R session. Now to see the

mode and class of each object, we can easily use the `mode()` and `class()` function.

To import a Stata file into R, we need to call the foreign library and then use the `read.dta()` function. Similarly, if we want to import an `SPSS` data file, the corresponding command will be `read.spss()`; the output will always be a data frame. Here is an example of importing a Stata file:

```
library(foreign)
iris_stata <- read.dta("iris_stata.dta")
```

In this section, we saw that a dataset can be stored in different formats and R has some user friendly functionality to deal with each of them. The noticeable feature of this section is some of the arguments within the `read.csv()` function, such as `skip`, `stringsAsFactors`, and `sep`. To import any data correctly, we have to use these arguments carefully.

# Factor manipulation

A variable that takes only a limited number of distinct values is usually known as a **categorical variable**, and in R, this is known as a **factor**. During data analysis, sometimes the factor variable plays an important role, particularly in studying the relationship between two categorical variables. In this section, we will see some important aspects of factor manipulation. When a factor variable is first created, it stores all its levels along with the factor. But if we take any subset of that factor variable, it inherits all its levels from the original factor levels. This feature sometimes creates confusion in understanding the results. Let us now see an example of this feature.

We will firstly create a factor variable from the `datamanipulation` character string with the English alphabet in lowercase as levels. Each letter of this string represents a value of that factor variable. Then, we will display the data with the `table()` function, where we will see lots of zero frequency corresponding to the letters that did not appear in the factor variable, as shown in the following code. We then drop those levels that are not part of the original factor variable and will display the data again.

```
# creating an R object whose value is "datamanipulation"
char.obj <- "datamanipulation"

# creating a factor variable by extracting each single letter from# the character
string. To extract each single letter the substring() # function has been used. Note:
nchar() function gives number of # character count in a character type R object
factor.obj <-
factor(substring(char.obj,1:nchar(char.obj),1:nchar(char.obj)),levels=letters)

# Displaying levels of the factor variable
levels(factor.obj)
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u"
"v" "w" "x" "y" "z"

# Displaying the data using the table() function
table(factor.obj)
factor.obj
a b c d e f g h i j k l m n o p q r s t u v w x y z
4 0 0 1 0 0 0 0 2 0 0 1 1 2 1 1 0 0 0 2 1 0 0 0 0 0
```

Notice that there are only a few nonzero values in the table because the original factor variable does not have the entire alphabet as its value. Now we will drop the levels that do not appear in the original factor variable. To do so, we will create another factor variable from the original factor variable, as shown in the following code:

```
# re-creating factor variable from existing factor variable
factor.obj1 <- factor(factor.obj)

# Displaying levels of the new factor variable
levels(factor.obj1)
 [1] "a" "d" "i" "l" "m" "n" "o" "p" "t" "u"

# displaying data using table() function
table(factor.obj1)
factor.obj1
a d i l m n o p t u
4 1 2 1 1 2 1 1 2 1
```

The important feature to notice here is that we can drop unused factor levels by recreating factor variables from the original factor variable. This is most useful when we use a subset of a factor variable.

# Factors from numeric variables

Numeric variables are convenient during statistical analysis, but sometimes we need to create categorical (factor) variables from numeric variables. We can create a limited number of categories from a numeric variable using a

series of conditional statements, but this is not an efficient way to perform this operation. In R, `cut` is a generic command to create factor variables from numeric variables. In the following example, we will see how we can create factors from a numeric variable using a series of conditional statements. We will also use the `cut` command to perform the same task.

```
# creating a numeric variable by taking 100 random numbers
# from normal distribution
set.seed(1234) # setting seed to reproduce the example
numvar <- rnorm(100)

# creating factor variable with 5 distinct category

num2factor <- cut(numvar,breaks=5)
class(num2factor)
[1] "factor"
levels(num2factor)
[1] "(-2.35,-1.37]"  "(-1.37,-0.389]" "(-0.389,0.592]" "(0.592,1.57]"   "(1.57,2.55]"
table(num2factor)
num2factor
(-2.35,-1.37] (-1.37,-0.389] (-0.389,0.592]  (0.592,1.57]  (1.57,2.55]
            7            43             29            13            8
```

By default, the levels are produced using the actual range of values. Sometimes, the range of values are given a specific name for convenience. For example, the five categories of the preceding factor might be called the lowest group, lower-middle group, middle group, upper-middle group, and highest group, as shown in the following code:

```
# creating factor with given labels
num2factor <- cut(numvar,breaks=5,labels=c("lowest group","lower middle group", "middle
group", "upper middle", "highest group"))

# displaying the data is tabular form

data.frame(table(num2factor))
          num2factor Freq
1       lowest group    7
2 lower middle group   43
3       middle group   29
4       upper middle   13
5      highest group    8
# creating factor variable using conditional statement
num2factor <- factor(ifelse(numvar<=-1.37,1,ifelse(numvar<=-
0.389,2,ifelse(numvar<=0.592,3,ifelse(numvar<=1.57,4,5)))),labels=c("(-2.35,-1.37]", "
(-1.37,-0.389]", "(-0.389,0.592]", "(0.592,1.57]",   "(1.57,2.55]"))

# displaying data using table function
table(num2factor)
num2factor
 (-2.35,-1.37] (-1.37,-0.389] (-0.389,0.592] (0.592,1.57] (1.57,2.55]
             7            43             29            13           8
```

Once we have converted the numeric variable to the factor variable and discarded the numeric variable, we cannot go back to the original numeric variable if needed. Therefore, we should be careful when converting the numeric variable to the factor variable.

# Date processing

R can handle date variables in several ways. There are built-in R functions available to deal with date variables, and there are also some useful contributed packages available. The built-in R function `as.Date()` can handle only dates but not time, whereas the `chron` package, contributed by James and Hornik in 2008, can handle both date and time. However, it cannot deal with time zones. Using the `POSIXct` and `POSIXlt` class objects, we can deal with the inclusion of time zones. But there is another R package, `lubridate`, contributed by Grolemund and Wickham in 2011, that gives much more user friendly functionality to deal with date and time with time zone support. In this section, we will see how we can easily deal with date and time using the `lubridate` package and compare it with built-in R functions.

Like other statistical software, R also has a base date, and using that base date, R internally stores `date` objects. In R, dates are stored as the number of days elapsed since January 1, 1970. So if we convert any `date` object to its internal number, it will show the number of days. We can reformat the number into a date using the `date` class. The following are some examples:

```
# creating date object using built in as.Date() function
as.Date("1970-01-01")
[1] "1970-01-01"

# looking at the internal value of date object
as.numeric(as.Date("1970-01-01"))
[1] 0

# Second January 1970 is showing number of elapsed day is 1.
as.Date("1970-01-02")
[1] "1970-01-02"
as.numeric(as.Date("1970-01-02"))
[1] 1
```

Using the `as.Date()` function, we can easily create the `date` object; the typical format of the `date` object in this function is year, month, and then day. But we can also create a `date` object with other formats by specifying the format within the `as.Date()` function, as shown in the following example:

```
# creating date object specifying format of date
as.Date("Jan-01-1970",format="%b-%d-%Y")
[1] "1970-01-01"
```

Note that when specifying the format of the date, we have to give the format that is aligned with the input string. For the complete list of code that is used to specify date formats, users are directed to the help documentation of the `strptime` function. Users can access the complete list by just typing in `help(strptime)` in the R console.

The `lubridate` package provides intuitive functionality to deal with the `date` object in R. The following are some of the examples to create the `date` object using the `lubridate` package:

```
# loading lubridate package
library(lubridate)

# creating date object using mdy() function
mdy("Jan-01-1970")
"1970-01-01 UTC"
```

Note that the default time zone in the `mdy`, `dmy`, or `ymd` function is **Coordinated Universal Time** (**UTC**). One of the most interesting and important features of the `lubridate` package is that it can deal with heterogeneous formats. Heterogeneous formats means users can store date information in various ways; for example, `second chapter due on 2013, august, 24, first chapter submitted on 2013, 08, 18, or 2013 aug 23`. From this heterogeneous date, we can extract the valid `date` object that can be processed further within R using the `lubridate` package, as shown in the following code:

```
# creating heterogeneous date object
hetero_date <- c("second chapter due on 2013, august, 24", "first chapter submitted on
2013, 08, 18", "2013 aug 23")
# parsing the character date object and convert to valid date
ymd(hetero_date)
[1] "2013-08-24 UTC" "2013-08-18 UTC" "2013-08-23 UTC"
```

Although the `lubridate` package can handle heterogeneous dates, the sequence of year, month, and day should be similar across all values within the same object, otherwise during date extraction there will be a missing value that will be generated along with a warning message. For example, if we alter the last date to `23 aug 2013`, this will not get converted into a valid date, as shown in the following code:

```
hetero_date <- c("second chapter due on 2013, august, 24", "first chapter submitted on
2013, 08, 18", "23 aug 2013")
ymd(hetero_date)
[1] "2013-08-24 UTC" "2013-08-18 UTC" NA
Warning message:
1 failed to parse.
```

During the date manipulation, sometimes we need to change the month only within an existing R `date` object. The following is an example of doing this using the default R function and also using the `lubridate` package:

```
# Creating date object using based R functionality
date <- as.POSIXct("23-07-2013",format = "%d-%m-%Y", tz = "UTC")
date
[1] "2013-07-23 UTC"
# extracting month from the date object
as.numeric(format(date, "%m"))
[1] 7

# manipulating month by replacing month 7 to 8
date <- as.POSIXct(format(date,"%Y-8-%d"), tz = "UTC")
date
[1] "2013-08-23 UTC"

# The same operation is done using lubridate package
date <- dmy("23-07-2013")
date
[1] "2013-07-23 UTC"
month(date)
[1] 7
month(date) <- 8
date
[1] "2013-08-23 UTC"
```

In a dataset, the variable might have both date and time information and we need to round them to the nearest day or month. The following example shows the date-rounding functionality; this example also displays how to convert the time zone:

```
# accessing system date and time
current_time <- now()
current_time
[1] "2013-08-23 23:43:01 BDT"

# changing time zone to "GMT"
current_time_gmt <- with_tz(current_time,"GMT")
current_time_gmt
[1] "2013-08-23 17:43:01 GMT"

# rounding the date to nearest day
round_date(current_time_gmt,"day")
[1] "2013-08-24 GMT"

# rounding the date to nearest month
round_date(current_time_gmt,"month")
[1] "2013-09-01 GMT"

# rounding date to nearest year
round_date(current_time_gmt,"year")
[1] "2014-01-01 GMT"
```

In this section, we saw that dealing with dates using the `lubridate` package is really user friendly and intuitive.

# Character manipulation

In any statistical software, all the data is expected to be either numeric or at least a factor, but sometimes we have to work with character data. In the area of text mining, character or string manipulation is the most important. R has complete functionality to manipulate character (string) data for further analysis. Besides default R functionality, there is one contributed package to deal with character data, which is more user friendly and intuitive compared to the base R counterpart. Wickham developed the `stringr` package in 2010 to manipulate character data with some user friendly functions. In this section, we will introduce different functions and their counterparts in a table so that the readers are able to use the functions from the `stringr` package easily:

| **Base R functions** | **stringr functions** |
|---|---|
| `paste()`: This function is used to concatenate a vector of characters with a default separator as a space. | `str_c()`: This has a functionality similar to `paste()`, but it uses empty as the default separator. It also silently removes zero-length arguments. |
| `nchar()`: This returns the number of characters in a character string. For `NA`, it returns `2`, which is not expected. For example:<br>`nchar(c("x","y",NA))`<br>`[1] 1 1 2` | `str_length()`: This is the same as `nchar()`, but it preserves `NA`. For example:<br>`str_length(c("x","y",NA))`<br>`[1] 1 1 NA` |
| `substr()`: This extracts or replaces substrings in a character vector. | `str_sub()`: This is the equivalent of `substr()`, but it returns a zero-length vector if any of its inputs are of zero length. It also accepts negative positions, which are calculated from the left of the last character. The end position defaults to `-1`, which corresponds to the last character. |
| Unavailable | `str_dup()`: This is used to duplicate the characters within a string. |
| Unavailable | `str_trim()`: This is used to remove the leading and trailing whitespaces. |
| Unavailable | `str_pad()`: This is used to pad a string with extra whitespaces on the left, right, or both sides. |

Other than the functions listed in the preceding table, there are some other user friendly functions for pattern matching. Those functions are `str_detect`, `str_locate`, `str_extract`, `str_match`, `str_replace`, and so on. To get more details about these functions, readers should refer to *the stringr: modern, consistent string processing* paper, by Wickham, which can be found at http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf.

# Subscripting and subsetting

Subscripting and subsetting a dataset is an integral part of data manipulation. If we need to extract a smaller part of any R object (vector, data frame, matrix, or list) that contains more than one element, we need to use subscripts. Subscripting is an approach to access individual elements of an R object, for example, accessing particular element of a vector. Usually, numeric integers are used for subscripting, but logical vectors can also be used for the same purposes. In R, the subscript starts from 1, and if we specify any negative subscript, it omits that position from the source object.

The following is an example of an R vector with 10 elements and the effect of positive and negative subscripting:

```
# creating a 10 element vector
num10 <- c(3,2,5,3,9,6,7,9,2,3)
# accessing fifth element
num10[5]
[1] 9

# checking whether there is any value of num10 object greater # than 6
num10>6
[1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE

# keeping only values greater than 6
num10[num10>6]
[1] 9 7 9

# use of negative subscript removes first element "3"
num10[-1]
[1] 2 5 3 9 6 7 9 2 3
```

Note that the subscripted indexes are written within square brackets. For one-dimensional vectors, we use a single index to access elements, but for two-dimensional objects such as data frames or matrices, we have to use two-dimensional subscripts. In that case, we have to use double square brackets for indexing. The first index is for representing rows and the second is for representing columns; for example:

```
# creating a data frame with 2 variables
data_2variable <- data.frame(x1=c(2,3,4,5,6),x2=c(5,6,7,8,1))

# accessing only first row
data_2variable[1,]
  x1 x2
1  2  5

# accessing only first column
data_2variable[,1]
[1] 2 3 4 5 6

# accessing first row and first column
data_2variable[1,1]
[1] 2
```

Similar indexing is used for matrices. For the `list` object, the indexing is different than that of data frames or matrices. To get access to a `list` object, we have to use `[[]]` for indexing; for example, `[[1]]` to get the first element of a list. If the list is nested within another list, we need to use a series of double square brackets within double square brackets. The following example creates a `list` object and accesses its elements:

```
list_obj<- list(dat=data_2variable,vec.obj=c(1,2,3))
list_obj
$dat
  x1 x2
1  2  5
2  3  6
3  4  7
4  5  8
5  6  1

$vec.obj
[1] 1 2 3
# accessing second element of the list_obj objects
list_obj[[2]]
[1] 1 2 3
```

Now, if we want to get access to the individual elements of `list_obj[[2]]`, we have to use the following command:

```
list_obj[[2]][1]
[1] 1
```

If the `list` object is named, we can get access to the elements of that list using the name as follows:

```
# accessing dataset from the list object
list_obj$dat
x1 x2
1  2  5
2  3  6
3  4  7
4  5  8
5  6  1
```

Subsetting is just storing subscripted objects. Once we extract any subscripted R object and store it in another variable, the newly created object is the subset of the original variable.

# Summary

In this chapter, we have covered some of the special features that we need to consider during data acquisition. We also discussed the important aspect of factor manipulation, especially when subsetting a factor variable and how to remove unused factor levels. The processing of date variables was covered with the use of the `lubridate` package, with its user friendly and intuitive functions, and also string processing has been highlighted. The chapter ended with an explanation of the concepts of subscripting and subsetting. For more details on date processing and string manipulation, readers should refer to *the stringr: modern, consistent string processing* paper, by Wickham, which can be found at http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf., and the *Dates and Times Made Easy with lubridate* journal, by Grolemund and Wickham, which can be found at http://www.jstatsoft.org/v40/i03/paper.

In the next chapter, we will discuss data manipulation with the `plyr` package, where we will focus on the split-apply-combine strategy, a state-of-the-art approach in the group-wise data manipulation using R.

# Chapter 3. Data Manipulation Using plyr

In any data analysis task, a majority of the time is dedicated to data cleaning and preprocessing. It is considered that sometimes about 80 percent of the effort is devoted to data cleaning before conducting actual analysis (*Exploratory Data Mining and Data Cleaning*, *Dasu T.* and *Johnson T.*). Data manipulation is an integral part of data cleaning and analysis. For large sets of data, it is always preferable to perform the operation within subgroups of a dataset to speed up the process. In R, this type of data manipulation can be done with base functionality, but for large-scale data it requires a considerable amount of coding and eventually takes more processing time. In the case of large-scale data, we can split the dataset, perform the manipulation or analysis, and then combine it into a single output again. This type of split using default R is not efficient, and to overcome this limitation, Wickham, in 2011, developed an R package called **plyr**, in which he efficiently implemented the **split-apply-combine** strategy. This chapter starts with the concept of split-apply-combine and is followed by the different functions and utilities of the `plyr` package.

## The split-apply-combine strategy

Often, we require similar types of operations in different subgroups of a dataset, such as group-wise summarization, standardization, and statistical modeling. This type of task requires us to break up a big problem into manageable pieces, perform operations on each piece separately, and finally combine the output of each piece into a single piece of output (the paper *The Split-Apply-Combine Strategy for Data Analysis*, by Wickham, which can be found at http://www.jstatsoft.org/v40/i01/paper). To understand the split-apply-combine strategy intuitively, we could compare this with the **map-reduce** strategy for processing large amounts of data, recently popularized by Google. In the map-reduce strategy, the map step corresponds to split and apply and the reduce step consists of combining. The map-reduce approach is primarily designed to deal with a highly parallel environment where the work has been done by several hundreds or thousands of computers independently.

The split-apply-combine strategy creates an opportunity to see the similarities of problems across subgroups that were previously unconnected. This strategy can be used in many existing tools, such as the `GROUP BY` operation in SAS, PivotTable in MS Excel, and the SQL `GROUP BY` operator.

To explain the split-apply-combine strategy, we will use Fisher's iris data. This dataset contains the measurements in centimeters of these variables: sepal length and width, and petal length and width, for 50 flowers from each of the three species of iris. The species are *Iris setosa*, *Iris versicolor*, and *Iris virginica*. We want to calculate the mean of each variable and for each species separately. This can be done in different ways using a loop or without using one.

### Split-apply-combine without a loop

In this section, we will see an example of the split-apply-combine strategy without using a loop. The steps are as follows:

1. Split the iris dataset into three parts.
2. Remove the species name variable from the data.
3. Calculate the mean of each variable for the three different parts separately.
4. Combine the output into a single data frame.

The code for this is as follows:

```
# notice that during split step a negative 5 is used within the # code, this negative 5
has been used to discard fifth column of the # iris data that contains "species"
information and we do not need # that column to calculate mean.

iris.set <- iris[iris$Species=="setosa",-5]
iris.versi <- iris[iris$Species=="versicolor",-5]
iris.virg <- iris[iris$Species=="virginica",-5]

# calculating mean for each piece (The apply step)
mean.set <- colMeans(iris.set)
mean.versi <- colMeans(iris.versi)
mean.virg <- colMeans(iris.virg)

# combining the output (The combine step)
mean.iris <- rbind(mean.set,mean.versi,mean.virg)

# giving row names so that the output could be easily understood
rownames(mean.iris) <- c("setosa","versicolor","virginica")
```

# Split-apply-combine with a loop

The following example will calculate the same statistics as in the previous section, but this time we will perform this task using a loop. The steps are similar but the code is different. In each iteration, we will split the data for each species and calculate the mean for each variable and then combine the output into a single data frame, as shown in the following code:

```
# split-apply-combine using loop
# each iteration represents split
# mean calculation within each iteration represents apply step
# rbind command in each iteration represents combine step

mean.iris.loop <- NULL
for(species in unique(iris$Species))
{
iris_sub <- iris[iris$Species==species,]
column_means <- colMeans(iris_sub[,-5])
mean.iris.loop <- rbind(mean.iris.loop,column_means)
}

# giving row names so that the output could be easily understood
rownames(mean.iris.loop) <- unique(iris$Species)
```

An important fact to note in the split-apply-combine strategy is that each piece should be independent of the other. If the calculation in one piece is somehow dependent on the other, the split-apply-combine strategy will not work. This strategy is not applicable in running an average type of operation, where a current average is dependent on the previous one. This strategy is only applicable when the big problem can be broken up into smaller manageable pieces and we can perform the desired operation on each piece independently. For running average calculations, the split-apply-combine strategy is not suitable; we can use a loop instead. But if processing speed is a concern, we can write the code in some lower-level language such as C or ForTran.

# Utilities of plyr

The `plyr` package is a set of tools for a common set of problems. We want to split the big problem into smaller pieces, apply functions, and then combine all the outputs back together. The example we presented using the iris data is one where we applied this strategy. Though it is already possible to perform split-apply-combine operations with base R, such as the split and apply family of functions, `plyr` makes things much easier and intuitive with its consistent naming convention, various types of input-output processing, and built-in error recovery and informative messages. In general, `plyr` provides a replacement for the `for` loop. We do not need to replace the `for` loop just because it is slow, but we need to replace it to avoid extra, unimportant bookkeeping code. The following examples will clarify the need to replace a `for` loop with its `plyr` counterpart.

The mean calculation of each variable within each species in the `iris` dataset using the `for` loop in base R can be coded as follows:

```
mean.iris.loop <- NULL
for(species in unique(iris$Species))
{
  iris_sub <- iris[iris$Species==species,]
  column_means <- colMeans(iris_sub[,-5])
  mean.iris.loop <- rbind(mean.iris.loop,column_means)
}
rownames(mean.iris.loop) <- unique(iris$Species)

mean.iris.loop
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

The same mean calculation, but this time using the `plyr` package, can be coded as follows:

```
library (plyr)
ddply(iris,~Species,function(x) colMeans(x[,-which(colnames(x)=="Species")]))

mean.iris.loop
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

Note that we can easily perform the same calculation with very little code using the `plyr` package compared to the `for` loop in base R.

# Intuitive function names

To perform any kind of data processing, we need to know the type of input we have to provide and the expected format of output. In most R functions, it is difficult to understand from function names what type of input they accept and what the expected outputs are. Function names in `plyr` packages are much more intuitive and instructive about their input and output type. Each function is named according to the type of input it accepts and the type of output it produces. The first letter of the function name specifies the input and the second letter specifies the output type; `a` represents array, `d` represents data frame, `l` represents list, and `_` (underscore) represents the output discarded. For example, the function name `adply()` takes input as array and produces output as a data frame. The following table gives us a complete idea about function-naming conventions used in the `plyr` package:

| Input | Output |
|---|---|

| | Array | Data frame | List | Discarded |
|---|---|---|---|---|
| **Array** | `aaply` | `adply` | `alply` | `a_ply` |
| **Data frame** | `daply` | `ddply` | `dlply` | `d_ply` |
| **List** | `laply` | `ldply` | `llply` | `l_ply` |

We can see that there are three types of input and four types of output. Users can easily get an idea of the types of input and output from the function names. Another interesting feature is that we do not need to learn all the 12 functions; rather, it is sufficient to learn the three types of input and four types of output.

Other than the function names in the table, there are some special cases of operating on arrays that correspond to the `mapply()` function in base R. In base R, `mapply()` can take multiple inputs as separate arguments, whereas `a*ply()` takes only a single array argument. However, the separate argument in `mapply()` should be of the same length. The `mapply()` functions that are equivalent to `plyr` are `maply()`, `mdply()`, `mlply()`, and `m_ply()`.

Note that whenever a function name is written using a star symbol, such as `a*ply()`, it indicates that the common input is an array. The output can be in any format: array, data frame, or list. Optionally, the output can be discarded.

To explain the intuitive nature of the input and output, we will now provide an example using the `iris` data that we used in an earlier example. This time, we will use the `iris3` dataset; this is the same data, but stored in a three-dimensional array format. We will calculate the mean of each variable for each species as shown in the following code:

```
# class of iris3 dataset is array
class(iris3)
[1] "array"

# dimension of iris3 dataset
dim(iris3)
[1] 50  4  3
```

The following code snippet calculates the column mean for each species with the input as an array and the output as a data frame:

```
# Calculate column mean for each species and output will be data frame
iris_mean <- adply(iris3,3,colMeans)

class(iris_mean)
[1] "data.frame"

iris_mean
          X1 Sepal L. Sepal W. Petal L. Petal W.
1     Setosa    5.006    3.428    1.462    0.246
2 Versicolor    5.936    2.770    4.260    1.326
3  Virginica    6.588    2.974    5.552    2.0266
```

The following code snippet calculates the column mean for each species with the input as an array and the output as an array too:

```
# again we will calculate the mean but this time output will be an # array
iris_mean <- aaply(iris3,3,colMeans)
class(iris_mean)
[1] "matrix"

iris_mean

X1           Sepal L. Sepal W. Petal L. Petal W.
  Setosa        5.006    3.428    1.462    0.246
  Versicolor    5.936    2.770    4.260    1.326
  Virginica     6.588    2.974    5.552    2.026

# note that here the class is showing "matrix", since the output is a # two dimensional
array which represents matrix

# Now calculate mean again with output as list
iris_mean <- alply(iris3,3,colMeans)
class(iris_mean)
[1] "list"

iris_mean
$'1'
Sepal L. Sepal W. Petal L. Petal W.
   5.006    3.428    1.462    0.246

$'2'
Sepal L. Sepal W. Petal L. Petal W.
   5.936    2.770    4.260    1.326

$'3'
Sepal L. Sepal W. Petal L. Petal W.
   6.588    2.974    5.552    2.026

attr(,"split_type")
[1] "array"
attr(,"split_labels")
          X1
1     Setosa
2 Versicolor
3  Virginica
```

# Input and arguments

The functions in the `plyr` package accept various input objects: data frames, arrays, and lists. Each input object has its own rule to split the process. In this section, we will discuss input and arguments. The rules of splitting can be described shortly as follows:

- Arrays are sliced by dimension into lower dimensional pieces, and the corresponding common function is `a*ply()`, where the array is the common input and the output can be one among an array, data frame, or list.
- Data frames are sliced and subset by a combination of variables from that dataset. The corresponding common function is `d*ply()`, where the data frame is the common input and the output can be one among an array, data frame, or list.
- The elements of a list are processed separately, and the common function is `l*ply()`, where the common input is a list and the output can be an array, data frame, or list.

Depending on the input type, there are two or three main arguments for the common functions: `a*ply()`, `d*ply()`, and `l*ply()`. The following are the main arguments for these common functions:

- `a*ply(.data, .margins, .fun, ..., .progress = "none")`
- `d*ply(.data, .variables, .fun, ..., .progress = "none")`
- `l*ply(.data, .fun, ..., .progress = "none")`

The first argument, `.data`, is the input dataset that needs to be processed by splitting up, and the output will be combined from each split. The `.margins` or `.variables` argument specifies how the data should be split up into

smaller pieces. The `.fun` argument specifies the processing task; this can be any function that is applicable to each split of the input. If we omit the `.fun` argument, the input data is just converted to the output structure specified by the function. If we want to monitor the progress of the processing task, the `.progress` argument should be specified. It will not show the progress status by default.

In the following example, we will see what will happen if we do not specify the `.fun` argument in any function of the `plyr` package. If we give the input as an array and want the output as a data frame, but we haven't given a `.fun` argument, the `adply()` function will just convert the array object into a data frame. Here is an example:

```
# converting 3 dimensional array to a 2 dimensional data frame
iris_dat <- adply(iris3, .margins=3)
class(iris_dat)
[1] "data.frame"
str(iris_dat)
'data.frame':   150 obs. of  5 variables:
$ X1      : Factor w/ 3 levels "Setosa","Versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
$ Sepal L.: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal W.: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$ Petal L.: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
$ Petal W.: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

The `.margins` argument works in a similar manner to the `apply` function in base R. It does the following:

- Slices up a row by specifying `.margins = 1`
- Slices up a column by `.margins = 2`
- Slices up the individual cells by `.margins = c(1,2)`

The `.margins` argument works correspondingly for higher dimensions, with a combinatorial explosion in the number of possible ways to slice up the array.

# Comparing default R and plyr

In this section, we will compare code side by side to solve the same problem using both default R and `plyr`. Reusing the `iris3` data, we are now interested in producing five number summary statistics for each variable group by species. The five numbers will be minimum, mean, median, maximum, and standard deviation. The output will be a list of data frames.

To calculate the summaries for the five numbers, follow these steps:

1. Define a function that will calculate five number summary statistics for a given vector.
2. Produce the output of this function in a data frame object.
3. Apply this function in the `iris3` dataset using a `for` loop.
4. Apply the same function using the `apply()` function of the `plyr` package.

An example that explains the calculation of the five number summary statistics is as follows:

```
# Function to calculate five number summary
fivenum.summary <- function(x)
{
  results <-data.frame(min=apply(x,2,min),
  mean=apply(x,2,mean),
  median=apply(x,2,median),  max=apply(x,2,max),
  sd=apply(x,2,sd))
  return(results)
}
```

The following is how we calculate the summaries for the five numbers using a `for` loop with default R:

```
# initialize the output list object
all_stats <- list()

# the for loop will run for each species
for(i in 1:dim(iris3)[3])
{
  sub_data <- iris3[,,i]
  all_stat_species <- fivenum.summary(sub_data)
  all_stats[[i]] <-  all_stat_species
}

# class of the output object
class(all_stats)
[1] "list"

all_stats
[[1]]
         min  mean median max        sd
Sepal L. 4.3 5.006    5.0 5.8 0.3524897
Sepal W. 2.3 3.428    3.4 4.4 0.3790644
Petal L. 1.0 1.462    1.5 1.9 0.1736640
Petal W. 0.1 0.246    0.2 0.6 0.1053856

[[2]]
         min  mean median max        sd
Sepal L. 4.9 5.936   5.90 7.0 0.5161711
Sepal W. 2.0 2.770   2.80 3.4 0.3137983
Petal L. 3.0 4.260   4.35 5.1 0.4699110
Petal W. 1.0 1.326   1.30 1.8 0.1977527
[[3]]
         min  mean median max        sd
Sepal L. 4.9 6.588   6.50 7.9 0.6358796
Sepal W. 2.2 2.974   3.00 3.8 0.3224966
Petal L. 4.5 5.552   5.55 6.9 0.5518947
Petal W. 1.4 2.026   2.00 2.5 0.2746501
```

Let's calculate the same statistics, but this time using the `alply()` function from the `plyr` package:

```
all_stats <- alply(iris3,3,fivenum.summary)

class(all_stats)
[1] "list"

all_stats
$'1'
         min  mean median max        sd
Sepal L. 4.3 5.006    5.0 5.8 0.3524897
Sepal W. 2.3 3.428    3.4 4.4 0.3790644
Petal L. 1.0 1.462    1.5 1.9 0.1736640
Petal W. 0.1 0.246    0.2 0.6 0.1053856

$'2'
         min  mean median max        sd
Sepal L. 4.9 5.936   5.90 7.0 0.5161711
Sepal W. 2.0 2.770   2.80 3.4 0.3137983
Petal L. 3.0 4.260   4.35 5.1 0.4699110
Petal W. 1.0 1.326   1.30 1.8 0.1977527

$'3'
         min  mean median max        sd
Sepal L. 4.9 6.588   6.50 7.9 0.6358796
Sepal W. 2.2 2.974   3.00 3.8 0.3224966
Petal L. 4.5 5.552   5.55 6.9 0.5518947
Petal W. 1.4 2.026   2.00 2.5 0.2746501

attr(,"split_type")
[1] "array"
attr(,"split_labels")
          X1
1     Setosa
2 Versicolor
3  Virginica
```

# Multiargument functions

Sometimes, we have to deal with functions that take multiple arguments, and the values of each argument can come from a data frame, a list, or an array. The `plyr` package has intuitive and user friendly functions to work with multiargument functions. In this section, we will see an example of generating random numbers from normal distribution with various combinations of mean and standard deviation. The values of mean and standard deviation are stored in a data frame. Now, we will generate random numbers using default R functions, such as the `for` loop, and also using the `mlply()` function from the `plyr` package. The parameter combinations are given in the following table:

| **n** | **Mean** | **Standard deviation** |
|---|---|---|
| 25 | 0 | 1 |
| 50 | 2 | 1.5 |
| 100 | 3.5 | 2 |
| 200 | 2.5 | 5 |

| | | |
|---|---|---|
| 500 | 0.1 | 2 |

With these parameter combinations, we will generate normal random numbers using default R and `plyr` as shown in the following code:

```
# define parameter set
parameter.dat <-
data.frame(n=c(25,50,100,200,400),mean=c(0,2,3.5,2.5,0.1),sd=c(1,1.5,2,5,2))

# displaying parameter set

parameter.dat
    n mean  sd
1  25  0.0 1.0
2  50  2.0 1.5
3 100  3.5 2.0
4 200  2.5 5.0
5 400  0.1 2.0

# random normal variate generate using base R
# set seed to make the example reproducible
set.seed(12345)

# initialize blank list object to store the generated variable
dat <- list()
for(i in 1:nrow(parameter.dat))
{
 dat[[i]] <- rnorm(n=parameter.dat[i,1],
             mean=parameter.dat[i,2],sd=parameter.dat[i,3])
}

# estimating mean from the newly generated data
estmean <- lapply(dat,mean)
estmean
[[1]]
[1] -0.001177287

[[2]]
[1] 2.417842

[[3]]
[1] 3.667193

[[4]]
[1] 2.999662

[[5]]
[1] 0.1765926

# Performing same task as above but this time use plyr package

dat_plyr <- mlply(parameter.dat,rnorm)
estmean_plyr <- llply(dat_plyr,mean)
estmean_plyr
$'1'
[1] 0.4252469

$'2'
[1] 2.037528

$'3'
[1] 3.070231

$'4'
[1] 2.144276

$'5'
[1] 0.05399488
```

# Summary

In this chapter, we discussed the importance of the split-apply-combine strategy. We understood what the split-apply-combine strategy is and why it is important in data manipulations. The split-apply-combine strategy can be implemented using base R, but requires a large amount of code and is not memory or time efficient. To overcome this limitation, we discussed the `plyr` package, in which group-wise data manipulation can be implemented efficiently. The functions within `plyr` are intuitive and instructive in terms of input and output type. A large variety of data processing can be done using only a few functions with common input and various outputs. For further reading, an interested user can look up (at the paper *The Split-Apply-Combine Strategy for Data Analysis*, by Wickham, which can be found at http://www.jstatsoft.org/v40/i01/paper). In the upcoming chapter, we will learn about reshaping a dataset, which is another important aspect of group-wise data manipulation.

# Chapter 4. Reshaping 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 to implement some reorientation to perform certain types of analyses. Datasets layout could be long or wide. In long-layout, multiple rows represent a single subject's record, whereas in wide-layout, a single row represents a single subject's record. Statistical analysis sometimes requires wide data and sometimes long data, and in such cases, we need to be able to fluently and fluidly reshape the data to meet the requirements of statistical analysis. Data reshaping is just a rearrangement of the form of the data—it does not change the content of the dataset. In this chapter, we will show you different layouts of the same dataset and see how they can be transferred from one layout to another. This chapter mainly highlights the *melt* and *cast* paradigm of reshaping datasets, which is implemented in the `reshape` contributed package. Later on, this same package is reimplemented with a new name, `reshape2`, which is much more time and memory efficient (the *Reshaping Data with the reshape Package* paper, by Wickham, which can be found at http://www.jstatsoft.org/v21/i12/paper). In this chapter, we will discuss the layout of a dataset and how we can change the layout using the new paradigm of reshaping datasets with `melt` and `cast`. To run the example of this chapter, readers need to install both the `reshape` and `reshape2` packages.

## The typical layout of a dataset

A single dataset can be rearranged in many different ways, but before going into rearrangement, let's look back at how we usually perceive a dataset. Whenever we think about any dataset, we think of a two-dimensional arrangement where a row represents a subject's (a subject could be a person and is typically the respondent in a survey) information for all the variables in a dataset and a column represents the information for each characteristic for all subjects. This means rows indicate records and columns indicate variables, characteristics, or attributes. This is the typical layout of a dataset. In this arrangement, one or more variables might play a role as an identifier and others are measured characteristics. For the purpose of reshaping, we could group the variables into two groups: identifier variables and measured variables.

- **Identifier variables**: These help to identify the subject from whom we took information on different characteristics. Typically, identifier variables are qualitative in nature and take a limited number of unique values. In database terms, an identifier is termed as the primary key, and this can be a single variable or a composite of multiple variables.
- **Measured variables**: These are those characteristics whose information we took from a subject of interest. These can be qualitative, quantitative, or a mix of both.

### Long layout

In this layout, the dataset is arranged in such a way that a single subject's information is stored in multiple rows. We need a composite identification variable to identify a unique row. This type of layout is usually seen in a longitudinal dataset. The following is an example of this type of dataset:

| sid | exmterm | math | literature | language |
|---|---|---|---|---|
| 1 | 1 | 50 | 40 | 70 |
| 1 | 2 | 65 | 45 | 80 |
| | | | | |

| | | | | |
|---|---|---|---|---|
| 2 | 1 | 75 | 55 | 75 |
| 2 | 2 | 69 | 59 | 78 |

Notice that in the dataset we have repeated `sid`, but if we consider both `sid` and `exmterm`, each row can be identified uniquely. This layout is known as the long layout. The following is the R code to produce this data frame:

```
# Example of typical two dimensional data

# A demo dataset "students" with typical layout. This data # contains two students'
exam score of "math", "literature" # and "language" in different term exam.
students <- data.frame(sid=c(1,1,2,2),
exmterm=c(1,2,1,2),
math=c(50,65,75,69),
literature=c(40,45,55,59),
language=c(70,80,75,78))
students
  sid exmterm math literature language
1   1       1   50         40       70
2   1       2   65         45       80
3   2       1   75         55       75
4   2       2   69         59       78
```

# Wide layout

In this layout, each row represents all the information of a single subject. Usually, only one identification variable is enough to identify a unique subject, but a composite identification variable can be used. The main difference between wide and long layout is that the wide layout contains all the measured information in different columns. The following is the wide layout of the same data that we initially stored in long layout:

| sid | math.1 | literature.1 | language.1 | math.2 | literature.2 | language.2 |
|---|---|---|---|---|---|---|
| 1 | 50 | 40 | 70 | 65 | 45 | 80 |
| 2 | 75 | 55 | 75 | 69 | 59 | 78 |

Notice that in this layout, each row contains all the information corresponding to a single value of `sid`. This layout is known as the wide form. In a later section, we will see how we can convert long to wide and vice versa using R.

# The new layout of a dataset

In R, the layout of a dataset is known differently than the typical layout we discussed in previous section. This new layout of a dataset consists of only the identification variables and a value per variable. The identification variable identifies a subject along with which measured variable the value represents, which is the long layout in this paradigm. In this new paradigm, each row represents one observation of one variable. Interestingly, the typical long and wide layouts are both known as wide layout in this new paradigm. In the new paradigm, long data is also known as molten data and the process of producing molten data is known as melting from the wide layout. The difference between this new layout of the data and the typical layout is that it now contains only the ID `variable` and a new column, `value`, which represents the value of that observation. The following is an example of molten data that comes from the typical long layout:

| **sid** | **exmterm** | **variable** | **value** |
|---|---|---|---|
| 1 | 1 | math | 50 |
| 1 | 2 | math | 65 |
| 2 | 1 | math | 75 |
| 2 | 2 | math | 69 |
| 1 | 1 | literature | 40 |
| 1 | 2 | literature | 45 |
| 2 | 1 | literature | 55 |
| 2 | 2 | literature | 59 |
| 1 | 1 | language | 70 |
| 1 | 2 | language | 80 |
| 2 | 1 | language | 75 |
| 2 | 2 | language | 78 |

In this dataset, we see that each row contains all the information of one student, which is known as the wide data. The following is the R code to generate this molten data:

```
# Example of molten data
library(reshape)
molten_students <- melt.data.frame(students,id.vars=c("sid","exmterm"))
```

The `melt.data.frame` function converts the wide data to long (molten) form, and the new layout will contain only the identification variables along with two other columns named `variable` and `value`. In the new layout, each row contains the observation of a single variable, which is also known as the long form. The `variable` column represents the identification information along with what is being measured and the `value` column contains the measurement itself.

# Reshaping the dataset from the typical layout

In this section, we will see how we can convert a typical long layout to a typical wide layout and vice versa. To perform this conversion, we will use the built-in `reshape()` function, which takes several arguments, but we will use the following arguments:

- `data`: This argument specifies the dataset that we want to change the layout of.
- `direction`: This argument specifies whether the data is long or wide. Note that here long and wide indicates the typical layout.
- `idvar`: This argument specifies the identification variable. It could be a single variable or multiple variables.
- `timevar`: This argument specifies how many times the values of `idvar` repeats for each subject.

The following example converts the long layout to a wide layout of the students' data that was created earlier:

```
# Reshaping dataset using reshape function

wide_students <- reshape(students,direction="wide",idvar="sid",timevar="exmterm")

wide_students
sid math.1 literature.1 language.1 math.2 literature.2 language.2
   1     50           40         70     65           45         80
   2     75           55         75     69           59         78
```

After reshaping the data, we see that the rows contain each student's exam record. Now we will change the layout from wide to long using the same function:

```
# Now again reshape to long format
long_students <- reshape(wide_students,direction="long",idvar="id")
long_students
 sid exmterm math.1 literature.1 language.1
   1       1     50           40         70
   2       1     75           55         75
   1       2     65           45         80
   2       2     69           59         78
```

The limitation of this default `reshape` function is that it can only deal with the long and wide structures. In reality, data might contain multiple nested levels. To deal with complex data structures, the `reshape` function is not useful; we could use the `reshape` package instead.

# Reshaping the dataset with the reshape package

As we have seen, there are two different paradigms for defining the layout of a dataset. To change the layout of a dataset following the steps of a new paradigm, we need to use the `reshape` packages, where all the functions are implemented following the new layout. The main idea of the `reshape` package is melting a dataset and then casting it to a suitable layout. In the section *The new layout of a dataset*, we have mentioned melting a dataset and what it looks like. Just to recall, in molten data, each row represents a single observation of a single variable in the dataset. Also, it contains only the `identifier` variables and a `value` variable to represent what is being measured. In this section, we will discuss melting with more examples and casting with molten datasets.

## Melting data

In R, melting is a generic operation and can be applied to various data types, including data frames, arrays, and matrices. Though melting can be applied to different R objects, the most common use is melting a data frame. To perform melting operations using the `melt` function, we need to know what the identification variables are and what the measured variables in the original input dataset are.

If we do not specify the identification variables and measured variables, by default any factor variables are assumed as the ID variables and any numeric variables are assumed as measured variables. To avoid this ambiguous operation, it would be good to specify it explicitly. If we specify only one type of variable, either identification or measured, the function assumes the remaining variable is of the other category. For example, if we specify only the ID variables, the remaining variables will be considered as measured variables and vice versa. The following example will clarify these points:

```
# original data

students
  sid exmterm math literature language
1   1       1   50         40       70
2   1       2   65         45       80
3   2       1   75         55       75
4   2       2   69         59       78

# Melting by specifying both id and measured variables

melt(students,id=c("sid","exmterm"),
   measured=c("math","literature","language"))

 sid exmterm   variable value
1    1       1       math    50
2    1       2       math    65
3    2       1       math    75
4    2       2       math    69
5    1       1 literature    40
6    1       2 literature    45
7    2       1 literature    55
8    2       2 literature    59
9    1       1   language    70
10   1       2   language    80
11   2       1   language    75
12   2       2   language    78

# Melting by specifying only id variables

melt(students,id=c("sid","exmterm"))

   sid exmterm   variable value
1    1       1       math    50
2    1       2       math    65
3    2       1       math    75
4    2       2       math    69
5    1       1 literature    40
6    1       2 literature    45
7    2       1 literature    55
8    2       2 literature    59
9    1       1   language    70
10   1       2   language    80
11   2       1   language    75
12   2       2   language    78
```

In the melting process, the `melt` function does not assume the ID or measured variables; there could be any number of variables in any order, which gives flexibility to deal with the complex dataset. One important thing to note is that whenever we use the `melt` function, all the measured variables should be of the same type; that is, the measured variables should be either *numeric*, *factor*, *character*, or *date*.

## Missing values in molten data

There could be two types of missing values in practice: one is **sampling zero** (that is, no response) and the other is **structural missing**. The sampling zeros are explicitly coded and represented in the dataset, but the structural missing depends on the structure of the dataset. The structural missing is implicit in the dataset represented by the absence of a certain combination of the ID variable. If we change the structure of a dataset from nested to crossed, the implicit missing no longer exists in the data, rather it explicitly appears in the new structure and care should be taken to deal with that data. The following simple example taken from the *Reshaping Data with the reshape Package* paper*, by Wickham, which can be found at* http://www.jstatsoft.org/v21/i12/paper, clearly explains implicit and explicit missing in two different data structures.

Consider a dataset with two ID variables, `sex` (male or female) and `pregnant` (yes or no). When the variables are nested, the missing value "pregnant male" is represented by its absence in the dataset, as shown in the following table. However, in a crossed view, we need to add the explicit missing value as there will now be a cell that must

be filled with a value.

| Sex | Pregnant | Value |
|---|---|---|
| Male | No | 10 |
| Female | No | 14 |
| Female | Yes | 4 |

The cross view of this table can be represented as follows:

| Sex | Pregnant | Not Pregnant |
|---|---|---|
| Male | | 10 |
| Female | 4 | 14 |

To deal with the implicit missing value, it is good to use `na.rm=TRUE` with the `melt` function to remove the structural missing. If we do not specify `na.rm=TRUE` during melting, we have to specify this during data analysis.

# Casting molten data

Once we have molten data, we can rearrange it in any layout using the `cast` function from the `reshape` package. There are two main arguments required to cast molten data:

- `data`: This is the molten data that we want to reshape.
- `formula`: This is the casting formula to determine the layout of the output data; for example, which variable should go into columns and which should go into rows. If we do not specify a formula, the cast will return the classic data frame.

There are other argument options for performing certain types of operations, if required. The basic casting formula is `col_var_1+col_var_2 ~ row_var_1+ row_var_2`, which describes the variables to appear in columns and rows. The following example shows how the `cast` function works:

```
# Melting students data

molten_students <- melt(students,id.vars=c("sid","exmterm"))
molten_students
   sid exmterm   variable value
1    1       1       math    50
2    1       2       math    65
3    2       1       math    75
4    2       2       math    69
5    1       1 literature    40
6    1       2 literature    45
7    2       1 literature    55
8    2       2 literature    59
9    1       1   language    70
10   1       2   language    80
11   2       1   language    75
12   2       2   language    78
```

Now use the `cast` function to return to the original data structure by specifying both row and column variables, as follows:

```
cast(molten_students,sid+exmterm~variable)
  sid exmterm math literature language
1   1       1   50         40       70
2   1       2   65         45       80
3   2       1   75         55       75
4   2       2   69         59       78
```

The following is the same operation, but specifying only row variables:

```
cast(molten_students,...~variable)
  sid exmterm math literature language
1   1       1   50         40       70
2   1       2   65         45       80
3   2       1   75         55       75
4   2       2   69         59       78
```

We now rearrange the data in a way that `sid` is now a separate column for each student, as follows:

```
cast(molten_students,...~sid)
  exmterm   variable  1  2
1       1       math 50 75
2       1 literature 40 55
3       1   language 70 75
4       2       math 65 69
5       2 literature 45 59
6       2   language 80 78
```

We again rearrange the data in a way that `exmterm` is now a separate column for each term, as follows:

```
cast(molten_students,...~exmterm)
  sid   variable  1  2
1   1       math 50 65
2   1 literature 40 45
3   1   language 70 80
4   2       math 75 69
5   2 literature 55 59
6   2   language 75 78
```

## Note

Note that the column names of the last two examples are not valid column names because they contain numbers. It is a limitation of R. R cannot automatically label row or column names unambiguously, so we have

to be careful about column names during analysis.

# The reshape2 package

Though the `reshape` package has various functions to perform various tasks that cannot be done using built-in R functions, this package is slow. To make this more time and memory efficient, Wickham reimplemented this package and developed another package, `reshape2`. The reason behind the development of the new `reshape2` package is to keep the functionality of the original `reshape` package so that users do not get confused. Some important new features of the `reshape2` package are as follows:

- It is much better than the original `reshape` package in terms of memory and time efficiency
- It uses several functions instead of only the `cast` function
- The multidimensional marginal total can be calculated

# Summary

This chapter introduced the theoretical framework for reshaping a dataset. The limitations of conventional approaches were pointed out and the new paradigm of data layout was highlighted. In the new paradigm, only two functions allow users to rearrange datasets into various layouts as required. This chapter also discussed structural missing and sampling zero, and how to deal with those missing during the melting process. For faster and large data rearrangement, readers were redirected to the `reshape2` package. In the next chapter, we will discuss how R can be connected with databases and handle large-scale data.

# Chapter 5. R and Databases

We noticed earlier that a dataset can be stored in any format using different software as well as relational databases. Usually, large-scale datasets are stored in database software. In data mining and statistical learning, we need to process large-scale datasets. One of the major problems in R is memory usage. R is RAM intensive, and for that reason, the size of a dataset should be much smaller than its RAM. Also, one of the major drawbacks of R is its inability to deal with large datasets.

This chapter introduces how to deal with large datasets that are bigger than the computer's memory and dealing with a dataset by interacting with database software. In the first few sections, we describe how to interact with database software with **Open Database Connectivity** (**ODBC**) and import datasets. This chapter will present an example of memory issues and then describe ODBC using an example of MS Excel and MS Access, dealing with large datasets with specialized contributed R packages. This chapter ends with an introduction to data manipulation using SQL through the `sqldf` package.

The first are two examples demonstrating memory problems in R:

- The following example explains the memory limitation of a computer system. R stores everything in RAM, and a typical personal computer consists of limited RAM (depending on the computer's operating system, that is, 32-bit or 64-bit).

```
# Trying to create a vector of zero with length 2^32-1.
# Note that the RAM of the computer on we are generating # this example is 8 GB
with 64-bit Windows-7
# Professional edition. Processor core i5.

x <- rep(0, 2^31-1)
Error: cannot allocate vector of size 16.0 Gb
In addition: Warning messages:
1: Reached total allocation of 8078Mb: see help(memory.size)
2: Reached total allocation of 8078Mb: see help(memory.size)
3: Reached total allocation of 8078Mb: see help(memory.size)
4: Reached total allocation of 8078Mb: see help(memory.size)
```

- The preceding example clarifies that R cannot allocate a vector that has size larger than the RAM. Now we will see another example that is related to the maximally addressable range of different types of numbers. The maximum addressable range for integers is 231-1.

```
# Maximum addressable range of inter vector
as.integer(2^31-1)
[1] 2147483647

# If we try to assign a vector of length greater than # maximum addressable length
then that will produce NA

as.integer(2^31)
[1] NA
Warning message:
NAs introduced by coercion
```

# R and different databases

Before going on to discuss large-scale data handling using R, we will discuss how R can interact with database software through ODBC. There are two principal ways to connect to a database: the first uses the ODBC facility available on many computers and the second uses the `DBI` package of R along with a specialized package for the particular database needed to be accessed. If there is a specialized package available for a database, we may find that the corresponding DBI-based package gives better performance than the ODBC approach. On the other hand, if a database does not have a specialized package to access, using ODBC may be the only option.

# R and Excel

An Excel file can be imported into R using ODBC. We will now create an ODBC connection with an MS Excel file with the connection string `xlopen`.

To create an ODBC connection string with an MS Excel file, we need to open the control panel of the operating system and then open **Administrative Tools** and then choose ODBC. A dialog box will now appear. Click on the **Add...** button and select an appropriate ODBC driver and then locate the desired file and give a data source name. In our case, the data source name is `xlopen`. The name of the Excel file can be anything, and in our case the file name is `xlsxanscombe.xlsx`. The following R code will import the corresponding Excel file into the R environment:

```
# calling ODBC library into R
library(RODBC)

# creating connection with the database using odbc package.
# We created the connection following the steps outlined in the # preceding paragraph

xldb<- odbcConnect("xlopen")

# In the odbcConnect() function the minimum argument required
# is the ODBC connection string.

# Now the connection created, using that connection we will import data

xldata<- sqlFetch(xldb, "CSVanscombe")

# Note here that "CSVanscombe"is the Excel worksheet name.
```

We can use other packages to import an Excel file, but at the same time R has the facility to import data using the ODBC approach. To use the ODBC approach on an Excel file, we firstly need to create the connection string using the system administrator.

# R and MS Access

To import data from the MS Access database, the procedure is the same as with Excel. First, we need to create a connection string from the system administrator and then connect with the database from R using the `RODBC` package.

Let us consider the Access database containing three different tables: `coveragepage`, `questionnaire1`, and `questionnaire2`. The connection string to access this database is `accessdata`. The following command can be used to import all the three tables as separate data frames in R:

```
# calling odbc library
library(RODBC)

# connecting with database
access_con<- odbcConnect("accessdata")

# import separate table as separate R data frame
coverage_page<- sqlFetch(access_con, "coverpage")
ques1 <- sqlFetch(access_con, "questionnaire1")
ques2 <- sqlFetch(access_con, "questionnaire2")
```

Using MS Excel and MS Access, we can deal with fairly large datasets, but sometimes it so happens that the dataset is too large and handling with Excel or Access is difficult. Also, Excel cannot deal with relational databases. To overcome this limitation, R has another functionality, which we will discuss in the following sections.

# Relational databases in R

In this section, we will try to provide a concise overview of different packages in R for handling massive data and illustrate some of them.

A popular approach to dealing with bigger datasets is the use of SQL, a different programming language. It might not be difficult for someone to learn another programming language, but as we are dealing with and talking about using R, the community of R users try to develop specialized packages to deal with large datasets. Those contributed packages successfully create interfaces between R and different database software packages that use relational database management systems (RDBMS), such as MySQL (`RMySQL`), PostgreSQL (`RPgSQL`), and Oracle (`ROracle`). To get the full benefit of these specialized packages, we have to install third-party software, and one of the most popular packages is `RMySQL`. This package allows us to make connections between R and the MySQL server.

MySQL, which can deal with a mid-size, multi-platform RDBMS is a popular software in the open source community. Some of its advantages include high-performance, being open source, and being free for non-commercial use. In order to install this package properly, we need to download both the MySQL server and `RMySQL`.

There are several R packages available that allow direct interactions with large datasets within R, such as `filehash`, `ff`, and `bigmemory`. The idea is to avoid loading the whole dataset into memory.

## The filehash package

The `filehash` package, which is used for solving large-data problems, was contributed by Roger Peng (The *Interacting with Data using the filehash Package for R* paper, available at http://cran.r-project.org/web/packages/filehash/vignettes/filehash.pdf). The idea behind the development of this package was to avoid loading the dataset into a computer's virtual memory. We must rather dump the large dataset into the hard drive and then assign an environment name for the dumped objects. Once a dataset is dumped into the hard drive, we can access the data using the assigned environment. In this way, we can deal with larger datasets and avoid the use of the computer's virtual memory and allow faster data manipulation. We will now discuss the basic steps of using this package through some examples.

Firstly, create a database that can be accessed later on. To create a database, we have to use the `dbCreate` function, which needs to be initialized (via `dbInit`) in order to be accessed, as shown in the following code. The `dbInit` function returns an S4 object that inherits from the `filehash` class.

```
library(filehash)
dbCreate("exampledb")
filehash_db<- dbInit("exampledb")
```

The primary interface of `filehash` databases consists of the functions `dbFetch`, `dbInsert`, `dbExists`, `dbList`, and `dbDelete`. All of these functions are generic in nature and specific methods exist for the database that work in the backend. The first argument that is taken by the functions within this package is an object of the `filehash` class. To insert some data into the database, we can simply call `dbInsert`. We retrieve those data values with `dbFetch`, as shown in the following code:

```
dbInsert(filehash_db, "xx", rnorm(50))
value<- dbFetch(filehash_db, "xx")
summary(value)
```

The `dbList` function lists all of the keys that are available in the database, the `dbExists` function tests to see if a given key is in the database, and the `dbDelete` function deletes a key-value pair from the database, as shown in the following code:

```
dbInsert(filehash_db, "y", 4709)
dbDelete(filehash_db, "xx")
dbList(filehash_db)
dbExists(filehash_db, "xx")
```

There is another very useful command, `dbLoad()`, that works in a similar way to the `attach()` function. Using the `filehash` package, the objects are attached but stored on the local hard disk. We may also assess the objects in the `filehash` database using the usual standard R subset and accessor functions such as `$`, `[[`, and `[`, as shown in the following code:

```
filehash_db$x<- runif(100)
summary(filehash_db$x)
summary(filehash_db[["x"]])
filehash_db$y<- rnorm(100, 2)
dbList(filehash_db)
```

After initializing a database using the default `DB1` format, it opens a file connection for reading and writing to the database file on the disk. This file connection will remain open until the database is closed via `dbDisconnect` or the database object in R is removed. There is a limit on the number of file connections that can be open at the same time, so to protect any database from unexpected results, we need to make sure the file connections are closed properly.

Just like `save.image` in base R, there are some utilities included in the `filehash` package and two of them are `dumpObjects` and `dumpImage`. The `dumpObjects` utility saves an object into the `filehash` database so that it can be accessed in the future if required. It does not save objects into R itself, which allows faster processing. Similarly, `dumpImage` saves the entire workspace to a `filehash` database. The `dumpList` function takes a list and creates a `filehash` database with values from the list. The list must have a non-empty name for every element in order for `dumpList` to succeed. The `dumpDF` utility creates a `filehash` database from a data frame where each column of the data frame is an element in the database. Essentially, `dumpDF` converts the data frame to a list and then calls `dumpList`. The following example shows how we can use `dumpDF`:

```
dumpDF(read.table("anscombe.txt", header=T), dbName="massivedata")
massive_environment<- db2env(db="massivedata")
```

The first element of `dumpDF()` is a data object. R will read the data within `dumpDF()`, so its memory does not have a copy of it. Space saved! So now our pretended large dataset, `anscombe.txt`, can be accessed through the `env01` environment. To access it, we use `with()`. Suppose we want to perform a linear regression of `y` on `x` and access the data using the variable names. In such cases, if you assign an object name for the `read.table` command, the memory will have a copy of the data, which is not desirable. Using the `with()` function, we can fit a model or compute summary statistics as usual as follows:

```
fit<- with(massive_environment, lm(Y1~X1))
with(massive_environment, summary(Y1))
with(massive_environment, Y1[1] <- 99))
```

# The ff package

As we have seen in the example in the introductory section of this chapter, R can only address objects that fit within the memory limits of its RAM and the maximally addressable range of 231-1 bytes. To overcome this limitation, Adler and Glaser, in 2010, developed the `ff` package. This package extends the R system and stores data in the form of native binary flat files in persistent storage such as hard disks, CDs, or DVDs rather than in the RAM. This package enables users to work on several large datasets simultaneously. It also allows the allocation of vectors or arrays that are larger than the RAM. The package comprises of two parts: one is the low-level layer written in C++ and the other is the high-level layer in R. This package is designed for convenient access to large datasets.

As users will only deal with the high-level layer, the following are the tasks we do in this layer:

- **Opening/creating flat files**: There are two basic functions, `ff` and `ffm`, to deal with opening and creating flat files. If we specify the `length` argument or the `dim` argument, a new file is created, otherwise R will open an existing file.
- **I/O operations**: These operations are controlled by the `[]` (for reading) and the `[] <-` (for writing) operators.
- **Generic functions and methods for the ff and ffm objects**: Methods for `dim` and `length` are provided and the `sample` function is converted to a generic function.

The primary argument for the functions `ff` and `ffm` require a filename in the `file` argument to specify the flat file. Whenever `length` (for `ff`) or `dim` (for `ffm`) is specified, as shown in the following code, a new flat file is created, otherwise an existing file is opened:

```
# A flat file with a length 10 is created
library(ff)
file1 <- ff(filename="file1", length=10,vmode="double")
str(file1)

list()
 - attr(*, "physical")=Class 'ff_pointer' <externalptr>
  ..- attr(*, "vmode")= chr "double"
  ..- attr(*, "maxlength")= int 10
  ..- attr(*, "pattern")= chr "/"
  ..- attr(*, "filename")= chr "D:/Book on R/Writing/outline/data_ch2/file1"
  ..- attr(*, "pagesize")= int 65536
  ..- attr(*, "finalizer")= chr "close"
  ..- attr(*, "finonexit")= logi TRUE
  ..- attr(*, "readonly")= logi FALSE
  ..- attr(*, "caching")= chr "mmnoflush"
 - attr(*, "virtual")= list()
  ..- attr(*, "Length")= int 10
  ..- attr(*, "Symmetric")= logi FALSE
 - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
```

The entries of `file1` can be modified with the `[]<-` operator. For example, the first 10 entries of the `rivers` dataset that contains the length of the 141 rivers in North America can be stored in an `ff` object, as shown in the following code:

```
# calling rivers data
data(rivers)
file1[1:10] <- rivers[1:10]

# Note that here file1 is an ff object whereas
# file1[...] returns default R vector

str(file1)
```

If required, we can perform sampling on the `ff` objects as follows:

```
# set seed to reproduce the example
set.seed(1337)
sample(file1,5,replace=FALSE)

[1] 735 392 524 450 600
```

Flat file objects are referenced when forming R objects using external pointers. In order to clear the references, the garbage collector, `gc()`, can be used as follows:

```
gc()
```

Calling `gc()` clears the reference to the file, but does not delete the file from the hard drive. Since the data is still

present, the flat file can be opened again at a later stage.

# R and sqldf

The `sqldf` package is an R package that allows users to run SQL statements within R. SQL is the popular programming language for manipulating data from relational databases, and the `sqldf` package creates an opportunity to work directly with SQL statements on an R data frame. With this package, the user can do the following tasks easily:

- Write alternate syntax for data frame manipulation, particularly for purposes of faster processing, since using `sqldf` (with SQLite as the underlying database) is often faster compared to performing the same manipulations in built-in R functions
- Read portions of large files into R without reading the entire file

The user need not perform the following tasks once they use `sqldf` because these are automatically done:

- Database setup
- Writing the `create table` statement, which defines each table
- Importing and exporting to and from the database
- The coercing of the returned columns to the appropriate class in common cases

# Data manipulation using sqldf

We can perform any type of data manipulation to an R data frame either in memory or during import. The following example shows the selection of a portion of the `iris` dataset using the `sqldf` package:

```
# Selecting the rows from iris dataset where sepal length > 2.5
# and store that in subiris data frame

library(sqldf)
subiris<- sqldf("select * from iris where Sepal_Width> 3")
head(subiris)

   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.7         3.2          1.3         0.2  setosa
3           4.6         3.1          1.5         0.2  setosa
4           5.0         3.6          1.4         0.2  setosa
5           5.4         3.9          1.7         0.4  setosa
6           4.6         3.4          1.4         0.3  setosa

nrow(subiris)
[1] 67
```

We can also select a smaller number of columns while filtering out some of the rows with a specified condition. The following example selects only sepal length, petal length, and species; however, this time, rows are filtered by values for petal length greater than 1.4:

```
subiris2<-
sqldf("select Sepal_Length,Petal_Length,Species from iris where Petal_Length> 1.4")

nrow(subiris2)
[1] 126

head(subiris2)
  Sepal_Length   Petal_Length    Species
1           4.6          1.5     setosa
2           5.4          1.7     setosa
3           5.0          1.5     setosa
4           4.9          1.5     setosa
5           5.4          1.5     setosa
6           4.8          1.6     setosa
```

If the dataset is too large and cannot entirely be read into the R environment, we can import a portion of that dataset using `sqldf`. The following example shows how we can import a portion of a `csv` file using the `sqldf` functionality. We will use the `read.csv.sql()` function to perform this task. This is an interface to `sqldf` that works like `read.csv` in R, except that it also provides a `sql=` argument. Not all of the other arguments of `read.csv` are supported.

In the following example, we will import the `iris.csv` file. We will import only sepal width and petal width along with the species information where petal width is greater than 0.4:

```
iriscsv<-read.csv.sql("iris.csv",sql="select Sepal_Width,Petal_Width,Species from file
where Petal_Width>0.4")

head(iriscsv)
     Sepal_Width Petal_Width      Species
1          3.3         0.5       "setosa"
2          3.5         0.6       "setosa"
3          3.2         1.4   "versicolor"
4          3.2         1.5   "versicolor"
5          3.1         1.5   "versicolor"
6          2.3         1.3   "versicolor"
```

An important thing to note is that in the original `iris.csv` file, the variable names were dot separated, but when we pass a SQL statement, we need to use an underscore as the variable name, otherwise it will output an error as follows:

```
iriscsv<-read.csv.sql("iris.csv",sql="select Sepal.Width,Petal.Width,Species from file
where Petal.Width>0.4")

Error in sqliteExecStatement(con, statement, bind.data) :
  RS-DBI driver: (error in statement: no such column: Sepal.Width)
```

We sometimes need to draw a random sample from a dataset but the original data file might be too large. In the following example, we will show how we can draw a random sample size of 10 from the iris data that is stored in the `iris.csv` file:

```
iris_sample<-
read.csv.sql("iris.csv",sql="select * from file order by random(*) limit 10")

iris_sample
     Sepal_Length Sepal_Width Petal_Length Petal_Width    Species
1             6.5         3.0          5.2         2.0  "virginica"
2             5.0         3.5          1.3         0.3     "setosa"
3             6.0         2.2          4.0         1.0 "versicolor"
4             6.9         3.1          5.4         2.1  "virginica"
5             6.2         2.8          4.8         1.8  "virginica"
6             5.1         3.8          1.9         0.4     "setosa"
7             5.8         2.6          4.0         1.2 "versicolor"
8             5.9         3.2          4.8         1.8 "versicolor"
9             6.4         2.9          4.3         1.3 "versicolor"
10            6.4         3.1          5.5         1.8  "virginica"
```

We can perform group-wise processing and aggregation using `sqldf`, which is a faster alternative to the `aggregate` function. For example, if we want to calculate the mean of each variable in the iris data for each species, the following is the code:

```
# Calculate group wise mean from iris data
iris_avg<-sqldf("select Species,
avg(Sepal_Length),avg(Sepal_Width),avg(Petal_Length),avg(Petal_Width) from iris group
by Species")

colnames(iris_avg) <- c("Species","Sepal_L","Sepal_W","Petal_L","Petal_W")

iris_avg
     Species Sepal_L Sepal_W Petal_L Petal_W
1     setosa   5.006   3.428   1.462   0.246
2 versicolor   5.936   2.770   4.260   1.326
3  virginica   6.588   2.974   5.552   2.026
```

The base R counterpart for performing the same operation is as follows:

```
aggregate(iris[,-5],list(iris$Species),mean)

     Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

Though both functions give us the same results, for larger datasets, `sqldf` is much faster than base R.

# Summary

At the beginning of this chapter, we showed you how we can deal with an MS Excel file as a database and how an MS Access database table can be imported into R. One of the major problems in R is that its memory is bound by the system virtual memory, and that is why the data should be smaller in size than the memory of a dataset to be able to work with it. But in reality, datasets are often larger than the virtual memory and sometimes the length of the array or vector exceeds the maximum addressable range. To overcome these two limitations, R can be utilized with relational databases. Contributed R packages exist to help in dealing with such large datasets, and they have been highlighted in this chapter, particularly `filehash` and `ff`. We also discussed `sqldf` for faster data manipulation.

# Appendix A. Bibliography

The citations for this book are as follows:


- *Exploratory Data Mining and Data Cleaning*, *Dasu T. and Johnson T.*, available at http://as.wiley.com/WileyCDA/WileyTitle/productCd-0471268518.html
- *Interacting with Data using the filehash Package for R*, *Peng R.*, available at http://cran.r-project.org/web/packages/filehash/vignettes/filehash.pdf
- *The Reshaping Data with the reshape Package*, *Wickham H.*, *Journal of Statistical Software*, *Volume 21*, *Issue 12*, available at http://www.jstatsoft.org/v21/i12/paper
- *chron: Chronological objects which can handle dates and times*, *James D. and Hornik K.*, *R package Version 2.3-44*, available at http://cran.r-project.org/web/packages/chron/index.html
- *stringr: modern, consistent string processing*, *Wickham H.*, available at http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf
- *Dates and Times Made Easy with lubridate*, *Grolemund G. and Wickham H.*, *Journal of Statistical Software*, *Volume 40*, *Issue 3*, available at http://www.jstatsoft.org/v40/i03/paper
- *The Split-Apply-Combine Strategy for Data Analysis*, *Wickham H.*, *Journal of Statistical Software*, *Volume 40*, *Issue 1*, available at http://www.jstatsoft.org/v40/i01/paper
- *The State of Naming Conventions in R*, *Bååth R.*, *The R Journal Vol. 4/2*, *December 2012*, available at http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf
- *ff: memory-efficient storage of large data on disk and fast access functions*, *Adler D. et al.*, *R package Version 2.2-12*, available at http://cran.r-project.org/web/packages/ff/index.html

# Index

## A



## B



## C



## D

# M



# N



# O



# P



# R

## S

## T



## W