Data Manipulation with R

Second Edition

Efficiently perform data manipulation using the split-apply-combine strategy in R

Jaynal Abedin

Kishor Kumar Das

Data Manipulation with R
Second Edition

Efficiently perform data manipulation using the
split-apply-combine strategy in R

Jaynal Abedin

Kishor Kumar Das

[PACKT] open source ✳
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Data Manipulation with R
Second Edition

Copyright © 2015 Packt Publishing

First published: January 2014

Second edition: March 2015

Credits

Authors
Jaynal Abedin

Kishor Kumar Das

Reviewers
Erik M. Rodríguez Pacheco

Dr. Abbass Ismail Sharif

Dr. Brian J. Spiering

Jitendra Kumar Yadav

Commissioning Editor
Veena Pagare

Acquisition Editor
Sonali Vernekar

Content Development Editor
Manasi Pandire

Technical Editor
Utkarsha S. Kadam

Copy Editors
Khushnum Mistry

Karuna Narayanan

Vikrant Phadke

Adithi Shetty

Project Coordinator
Leena Purkait

Proofreaders
Stephen Copestake

Maria Gould

Paul Hindle

Jonathan Todd

Indexer
Monica Ajmera Mehta

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Authors

Jaynal Abedin currently holds the position of senior statistician at the Centre for Communicable Diseases (CCD) at the International Centre for Diarrhoeal Disease Research, Bangladesh (`http://www.icddrb.org/`). He attained his bachelor's and master's degrees in statistics from the University of Rajshahi, Bangladesh. He has extensive experience in R programming and Stata, and has good leadership qualities. He has contributed to two books on R and also developed an R package named edeR, short for e-mail data extraction using R, which is available at CRAN (`http://cran.r-project.org/web/packages/edeR/index.html`). 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 areas of interest in research include causal inference and machine learning. He is currently involved in several ongoing public health research projects, and is a coauthor of nine peer-reviewed scientific papers. Moreover, he is involved in several work-in-progress manuscripts. He works as a freelance statistician in online marketplaces and has obtained a good reputation for his work.

Kishor Kumar Das is a statistician at the International Centre for Diarrhoeal Disease Research, Bangladesh, an internationally recognized organization that focuses mainly on public health research. He completed his MSc and BSc in applied statistics from the Institute of Statistical Research and Training, University of Dhaka, Bangladesh. He has extensively used R for data processing, statistical analysis, and graphs for more than 10 years. His research interests are survival analysis, machine learning, and statistical computing.

About the Reviewers

Erik M. Rodríguez Pacheco works as a manager in the business intelligence unit at Banco Improsa in San José, Costa Rica. He has 11 years of experience in the finance industry. He is currently a professor of the Business Intelligence Specialization program at the Continuing Education Programs of Instituto Tecnológico de Costa Rica. Erik is an enthusiast of new technologies, particularly those related to business intelligence, data mining, and data science. He holds a bachelor's degree in business administration from Universidad de Costa Rica, a specialization in business intelligence from Instituto Tecnológico de Costa Rica, a specialization in data mining from Promidat (Programa Iberoamericano de Formación en Minería de Datos), and a specialization in business intelligence and data mining from Universidad del Bosque, Colombia. He is currently enrolled in a specialization program in data science from Johns Hopkins University. He can be reached at `http://cr.linkedin.com/in/erikrodriguezp`.

Dr. Abbass Ismail Sharif is an assistant professor of clinical data sciences and operations at the University of Southern California. He holds a PhD in statistics, an MS in computer science, and an MS in instructional technology and learning sciences. Abbass does research in the field of statistical computing and data visualization. For this purpose, he extensively uses the R statistical environment. He has developed new multivariate visualization techniques for functional data, and is currently developing visualization techniques to study brain activity data collected using Near-infrared spectroscopy (NIRS) technology.

Abbass has won a prestigious research award from the American Statistical Society for his doctoral work. He teaches both graduate and undergraduate statistics courses that range from introductory statistics and data analysis for decision-making to advanced modern statistical learning techniques, statistical computing, and data visualization.

Dr. Brian J. Spiering started coding in his elementary school's computer laboratory, hacking BASIC to make programs that entertained his peers and annoyed the school authorities. Much later, he earned a PhD in psychology from the University of California, Santa Barbara, with emphasis on cognition, perception, and cognitive neuroscience. His research is focused on building mathematical and computer models of the human brain and behavior. He has taught biological psychology, data analysis, and statistics. Brian currently works as a data scientist and resides in San Francisco, California, USA.

Jitendra Kumar Yadav is a senior development architect working in research and development for product development and innovation. He is an expert in cloud and big data product development. He has contributed to the open source community in the form of code development and support for a variety of platforms based on big data, cloud technologies, virtualization, storage, networking, and cloud security. For this, he has used programming languages such as C++, Python, R, Java, Go, and Perl.

Jitendra loves to share his knowledge with fellow techies and others. He does so by publishing papers and books and attending corporate tech events. He has won several awards for his excellent contributions to product development in the fields of cloud computing, big data, artificial intelligence, and virtualization. He has over 12 years of professional experience, and has spent most of his time in research and development.

Occasionally, when Jitendra needs to take a break, he spends his time traveling.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

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, you will be able to efficiently manage and check the validity of your datasets with the effective use of R programming, including specialized packages for data management. You will come to know about the split-apply-combine strategy, which is a state-of-the-art approach in data management. You will also come to know the way to work with database software through ODBC with the help of very simple examples. This book ends with an introduction to text processing for text mining using R.

What this book covers

Chapter 1, *Introduction to R Data Types and Basic Operations*, discusses the way to get R, how to install it, and how to install various libraries. Upon introducing how to write commands in R, this chapter discusses 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 as well as their modes and classes. 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 work 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 characters 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 textbooks—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 where we need to take care during data acquisition. Then, an important aspect of factor manipulation is discussed, as well as subsetting a factor variable and how to remove unused factor levels. This chapter also includes coverage of vector and matrix operations. Date processing has been discussed using an efficient R package: lubridate. Working with the date variable using the lubridate package is much more efficient than using any other existing package that is designed to work with the date variable. Also, string processing has been highlighted, and the chapter ends with a description of subscripting and subsetting.

Chapter 3, *Data Manipulation Using plyr and dplyr*, 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 a large dataset, it is always preferable to perform 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 datasets, it requires a considerable amount of coding and eventually takes longer to process. In the case of large datasets, we can split the dataset performing the manipulation or analysis and then combine them again into a single output. This chapter contains a discussion of the different functions in the plyr package that are used for group-wise data manipulation and also for data analysis. This chapter also contains examples and discussions of the dplyr package to work with data frames. Working with data frames using dplyr is much more efficient and intuitive. You will have a very good understanding of data frame processing through the examples of this chapter.

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 this case, we need to be able to fluently and fluidly reshape data to meet the requirements of statistical analysis. Important functions from the reshape2 package have been 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 the system virtual memory, and that is why working with a dataset requires the data to be smaller than its memory. However, in reality, the dataset is larger than the virtual memory 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 have been discussed with examples in this chapter.

Chapter 6, *Text Manipulation*, covers the processing of text data for text mining. This chapter introduces various sources of text data and the process of obtaining that data. This chapter also discusses processing text data for text mining purposes by using various relevant packages.

What you need for this book

Knowledge about statistical data is required. You are expected to have basic knowledge of R. 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 3.0.2.

Who this book is for

This book is for intermediate-to-advanced level users of R who have knowledge about datasets, and also for those who regularly work 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."

> Warnings or important notes appear in a box like this.

> 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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in 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 at `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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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.

1
Introduction to R Data Types and Basic Operations

R is an object-oriented programming language and an environment that is a variation of the S language written by Ross Ihaka and Robert Gentlemen (hence, the name R). 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 to perform specified tasks) to solve specific problems. We can develop up-to-date statistical techniques through R packages. Most importantly, R is open source and is a freely available software that will remain free.

Assuming that readers have very preliminary or no knowledge of R, the layout of this chapter is divided in to two major sections; the first one will be an introduction to R, and the second major section will relate to data types and basic operations.

The following are the reasons to use R:

- **R is free**: It comes with a license, but we do not have to pay anything to get it. It is not only free, but also open source. We can see the source code, change it as per our own requirements, and also distribute it without violating the license. Academicians across different disciplines around the world reviewed the core of the R system and also contributed to make it better.

- **R is a powerful software**: It is used to perform data processing and data analysis, and to produce a variety of graphs. All the necessary functions for data processing are available in R. It has a substantial collection of libraries (a library is a collection of functions to perform certain types of task), which are written by researchers working in a variety of fields. That is why, whether you are a statistician, biologist, environmentalist, or data scientist, you should find a set of functions that serves your purpose. The graphic system in R is one of the most powerful tools in this era. We have full control over every part of graphs produced in R.

- **R is up-to-date**: R is now one of the standard platforms to implement our research work. We should be able to find an R package suitable for the most recent developments, whatever our field is.

- **R is a community**: R is being developed by a team of volunteers. Also, it includes large communities that are writing new functions every day and that can help us out if we face any problem.

- **R is the language of communication**: R is now becoming a prominent way of sharing new findings with other researchers in this field.

Here is a summary of why we should use R:

- R is free, and it will remain free.
- It involves up-to-date implementation of recent statistical techniques.
- There is flexibility. The user has control over each and every part of a dataset and each component of each output.
- It is customizable based on the user's need.
- It has a large number of built-in libraries.
- It has a cloud-computing feature.
- It has rich graphics.
- It has a wide range of flexible data structures.
- It intelligently handles missing values.

Getting different versions of R

The source code, documentation, and other related files are maintained in the **Comprehensive R Archive Network** (**CRAN**), which can be found at `http://cran.r-project.org/`. CRAN is a collection of websites that contain identical materials consisting of the R distributions, contributed extensions, and documentation for R and binaries. The user can select anyone of the CRAN sites to download the R software. The user can download the software that is compatible to their computer's platform such as Windows, Mac, and Linux.

---

[2]

To download binaries for different platforms, anyone can use the following links:

- For Linux, `http://cran.at.r-project.org/bin/linux/`
- For Mac OS X, `http://cran.at.r-project.org/bin/macosx/`
- For Windows, `http://cran.at.r-project.org/bin/windows/`

The preceding links are applicable to download the most recent version of R. The latest R Version 3.1.2 (Pumpkin Helmet) was released on October 31, 2014.

To get the old version of R, Windows users can look at the various releases at `http://cran.r-project.org/bin/windows/base/old/`, and Mac users can look at `http://cran.r0-project.org/bin/macosx/old/` to download the desired one.

Installing R on different platforms

To install R on various platforms, the first requirement is to download appropriate binaries that are compatible with the relevant platform. In this section, we will briefly discuss installation on the Windows platform and will refer users to `http://cran.r-project.org/doc/manuals/r-release/R-admin.html` for the documentation for alternative platforms.

Installing R under Windows is as easy as installing any other software. After downloading the binary file for Windows (it comes with an `.exe` file), the name is for example, `R-3.1.2-win.exe`. This executable file contains binaries for a base distribution and a large number of add-on packages from CRAN. Users can install it just by double-clicking on the file and following the on-screen instructions. There is no special care that needs to be taken during installation; just go with the default selections.

Installing and using R libraries

R comes with a number of default packages, a collection of previously programmed functions for specific tasks, and with datasets. This is usually known as a library, but the R community refers to it as a package. There are two types of R packages:

- Default packages that come with the R executable
- Add-on packages that do not come during installation; we need to install them manually on downloading

When we open the R console, it automatically loads its default packages with the associated functions, and we do not need to load those packages manually. A list of installed packages can be obtained by typing `library()` in the R console. However, some of the packages need to load to execute functions. To load a specific package, the corresponding R command is `library(package)`, where `package` is the name of any library such as `plyr`, provided that the package has already been installed.

In some situations, we may require a special type of data processing and analysis. If the corresponding packages are not available in the default list, we need to install them. For example, the `plyr` package is not in the default list, so we need to install it separately.

There are two different ways to install a package:

- By manually downloading and installing it
- Installing it from within R

Manually downloading and installing packages

To download a package from CRAN and install it, follow these steps:

1. Go to `http://www.r-project.org/`.
2. Click on **CRAN mirror** under the **Getting Started** section.
3. Select any one of the regional servers from the list; for example, select the server from **Austria** at `http://cran.at.r-project.org/`.
4. Click on **Contributed extension packages** under the **Source Code for all Platforms** section.
5. Select **Table of available packages, sorted by date of publication** or **Table of available packages, sorted by name** and then download the desired package from the list.
6. While downloading, users need to choose the file that matches with the platform; for example, a Windows user will download the binary zip file.
7. Once the download is completed, open R.
8. Go to the **Packages** menu and select **Install packages from local zip files**.

One potential problem with manual downloads is that, sometimes, particular packages are dependent on other packages that are not included in the manual process of installation. To avoid this problem, we can install the desired package(s) from the R shell, as installing package(s) from the R shell resolves dependencies.

Installing packages within the R shell

To install a package from within the R console, we can use the `install.packages()` command; this command will prompt us to select the appropriate server CRAN. Note that to install packages using this approach, the computer must have active Internet connection.

For example, to install the `plyr` package, we can use the following command:

```
install.packages("plyr")
```

The previous command will prompt us to select a regional server and, after selecting the server from the available list, the package will be installed on the local computer.

Comparing R with other software

A growing number of libraries, currently more than 6,000, is the most noticeable feature of R, compared to other commercial software such as SAS, Stata, SPSS, and open source software such as Python and Octave. This feature enables R to have a huge number of tools for data management and statistical analysis. Data management capability is very limited in SPSS and Octave. The capability of R's data management is only comparable with commercial software such as SAS and open source software such as Python. R has no competitor that gets the most up-to-date packages for analysis in many areas such as finance, mathematics, data mining, machine learning, or even astronomy. Recently developed statistical analysis techniques are found in Python and Octave, but it took a while to get them in SPSS, Stata, and SAS.

R has a more intuitive syntax structure than the previously mentioned software. Its object-oriented features make it more flexible than SPSS, Stata, SAS, and Octave. Python shares the object-oriented features too, but it is less flexible than R. Open source software is designed to be developed by volunteer developers and offer very easy-to-implement function-writing capabilities. Although it is easy to write a function in Python and Octave, writing functions in R is even easier.

R has one of the best graphics systems among all existing software. The grammar graphics implemented in the `ggplot2` package makes it the most popular library for producing a variety of graphs with excellent quality. It is comparatively easy to modify all the components of a graph in R, compared to SPSS, Stata, SAS, Octave, and Python.

SPSS is very easy to use at first for some basic analysis, but when it comes to data management, scripting, and complex statistical analysis, sometimes it fails, and sometimes, it is very hard to implement. Learning Stata is very easy for basic data management tools, but if we want to do a complex data management function, it is very hard to implement. R has a very steep learning curve like Python, Octave, and SAS. However, unlike Octave and SAS, we can find a large number of freely available resources and tutorials on the Web to get help. These resources can make our learning easier compared to other software.

R as an enterprise solution

Revolution Analytics (`http://www.revolutionanalytics.com/`) is a statistical software company focused on developing open core versions of R, for enterprise, academic, and analytics customers. This type of enterprise solution supports big data analytics, various types of complex modeling of real-world problems, and day-to-day activities in big enterprises.

Writing commands in R

The R programming language is basically command-line (interpreter-type) programming. We can perform any type of mathematical and statistical calculation, including data management analysis and graphics in the command line. The `R` command window is known as the R console, where the command and the results are produced upon execution of a given command.

Here is a very basic example of using the R console:

```
> (44/55)*100
[1] 80
> log(25)
[1] 3.218876
> log10(25)
[1] 1.39794
> exp(0.23)
[1] 1.2586
```

```
> 453/365.25
[1] 1.240246
> 1-5*0.2
[1] 0
> 1-0.2-0.2-0.2-0.2-0.2 # An interesting calculation
[1] 5.551115e-17
```

Using the R console, we can perform any type of calculation, but we always need to preserve the code to reproduce the result of any scientific analysis. From this perspective, the R console is not user-friendly when it comes to saving commands. To save the necessary commands for future use and to ensure reproducibility of research results, R has a command editor, which is known as the script editor. The script editor is just like a plain text editor. We can preserve code and comments in R script files. The R console allows only one line of command at a time, and it executes as soon as we enter. However, in the script file, we can run a batch of code at a time. To write any type of comment related to any analysis in R, we can place a # (hash) sign as the starting character. Here is an example:

```
# This is a comment line
```

R data types and basic operations

In this major section of the chapter, we will introduce data types and structure and how to convert one type to another with very simple functions.

Modes and classes of R objects

Whatever we do in R, is stored 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 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 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
```

One important thing about objects in R is that, if we do not assign an object to any variable, we will not be able to reuse 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. Thus, whenever we deal with an object, we will assign it to an appropriate variable; 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 uppercase and lowercase, or even a combination of uppercase, lowercase, a number, and/or a dot. However, there are some rules to define variable names. For example, the name cannot start with numbers; it must start with a character or an 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 naming convention for functions is `lowerCamelCase`, while the most popular naming convention for arguments separates them by a period. 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 constructed 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 may 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"
```

[9]

```
> 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 a double-precision floating point.

If an R object contains both numeric and logical elements, the mode of that object will be numeric and, in this case, the logical element automatically gets converted to a numeric element. 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: 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. Here is an 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, 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 is 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 the two objects are printed using the print command is determined by its class. Here is an 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 factor class. This class allows less storage location, because it is required to store unique levels only once.

Interestingly, once we try to see the mode of a factor object, it always shows as numeric, even if it displays character data. Here is an 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 types* section in this chapter), a class is always a matrix or array, but its mode can be numeric, character, or logical.

The 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 are needed to store these 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 a multiple-element vector. The `num.obj` vector we created in the previous section is a kind of vector that comprises numeric elements.

One of the simplest ways to create a vector in R is to use the `c()` function. Here is an 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. Here is an 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 can be named, or it can be 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.

Here is an 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; these are matrix, data frame, and list. We will discuss each of these types in subsequent sections.

To convert the object mode, R has user-friendly functions that can be depicted as `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 numeric mode, and the data is stored in some other mode, the operation cannot be performed. In this case, we need to convert that data into 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 numeric mode to 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), where 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 offering any warning. However, if we want to convert character objects to any other type, we have to be careful.

Vector

R is a domain-specific programming language, specially designed to perform statistical analysis on data. In statistics, when we analyze data, the first thing that comes to mind is a variable with hundreds of observations in it. This reminds us of the picture of a vector. Probably, this is the main reason why, in R, the most elementary data type is a vector. A vector is a contiguous cell that contains data, where each cell can be accessed by an index:

```
> age <- c(10,20,30,40)
```

This is an example of a vector. The age of five individuals is stored in the age vector. Pay attention to how the vector was formed and stored under the `age` variable. Here, `c()` is a function used to create a vector, but this does not store all the data in the system. `<-` is called an assignment operator that is used to store a vector under a variable.

Now, in the console, let's type the following line and press *Enter*:

```
> age
 [1] 10 20 30 40
```

We successfully stored all the ages under the `age` variable, but what is `[1]`? This means that the index of the value `10` is `1`.

If you want to see the first values of the vector, type the following command:

```
> age[3]
[1] 30
```

Why did R only show the index of the first value and not the other values? This is only to keep the output clean and informative. Every time R writes a new line, it first gives the index number of the next value. Pretty soon, you will be familiar with this convention. We can store a single value under a variable, but it will be a vector with one element:

```
> height<- 175
```

To show you that height is not a scalar but a vector with one element, we will store one additional value in it:

```
> height[2]<- 180
```

---

[17]

Pay attention to how we added another value inside an existing vector. Here, we put 180 in the second cell of the vector. Can you recall how we accessed the value in the second cell for the age variable? Using `age[2]`, right? Similarly, we can assign a value to the second cell of the vector using the same syntax. Let's try to put another value inside the height variable:

```
> height[3] <- 165
```

Now, we can see all the values stored inside the height variable:

```
> height
[1] 175 180 165
```

Although the basic data structure in R is vectors, there can be different types of vector. We use a numeric vector to store numeric data such as age, height, weight, and so on. Character vectors are used to store string data such as name, address, and so on. The way we can define a character vector in R is simple:

```
> name<- c("Rob", "Bob", "Jude","Monica")
```

When we want to store a character in R, we need to use double quotes, as used in the previous example. This tells R that this is a string input. We can put numeric values using double quotes but, if we use a character without double quotes, then it will return an error message.

Another special type of vector is the logical vector. There are two ways we could define a logical vector; first, we will show you the more formal way and, second, we will show you the quick way. There can be two possible elements in a `logical` vector: `TRUE` and `FALSE`. This `logical` vector is used in `logical` operations in R. It can be used to select specific rows from a dataset.

We can define a `logical` vector in the following way:

```
> logical<- c(TRUE, FALSE, TRUE, FALSE)
```

This `logical` vector can be used as a row selector of the `age` vector in the following way:

```
> age[logical]
[1] 10 30
```

Look closely to find out what we just did. We have seen how we can extract age from a vector using indexing. A `logical` vector can be thought of as a vector of an index. The first element of the `logical` vector is TRUE, which means that the first element of the age vector will be selected. The second element of the `logical` vector is FALSE. This means that the second element of the age vector will not be selected. So, the `logical` vector will select only the elements of the age vector for which the `logical` vector is TRUE. So, finally, two elements of the age vector will be selected, and a vector of two elements will be returned. A question that may come to your mind is, What can we do with this feature? The answer will be clearer in the *Data frame* section.

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, 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 be returned 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 a label stores only once for each unique character. Factors can be ordered if the `ordered=T` command is specified; otherwise, they inherit 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. When we tried to perform a mathematical operation, it returned a warning message 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 first convert the factor into a character using `as.character()` and then use `as.numeric()`.

The following example describes this 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; it is rather a collection of variables. Each column represents only one type of data: numeric, character, or logical. 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 us 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 look at 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. However, 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"
# 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"
```

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:

- The data frame name followed by double square brackets with variable names within quotation marks–for example, `diab.dat[["var1"]]`
- The 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. When 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:

```
# To run the folloing code snipped,
# the code block 16 need to run.
# Especially var1 var2 var3 and var4.
```

```
# After that, from code block 17 "diab.dat.2" object should run

# The following line will remove var1 to var4
# object from the workspace
> rm(var1);rm(var2);rm(var3);rm(var4)
# The following command will allow
# us to access individual variables
> attach(diab.dat.2)
# Printing valsue of var1
> var1
# checking calss of var3
> class(var3)
# Now to detach the data frame from the workspace
> detach(diab.dat.2)
# Now if we try to print individual varaiable it will give error
> var1
```

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 operation, 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()`. However, this is not suitable for performing 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 can 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.

Here is a simple example to store three matrices of order 2 x 2 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 var1, var2, var3, and var4 vectors, 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=d
iab.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 can use the name of that element 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; alternatively, we can use this function in combination with the any() function.

The following example shows whether there is any missing value 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 first talked very briefly about what R is, where and how to get it, and how to install it. We then covered why we should use R and compared it with other available software. After that, we described what R objects are, 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, array, 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.

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 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:

- Acquiring data
- Vector and matrix operations
- Factor manipulations
- Factors from numeric variables
- Date processing using lubridate
- Character and string 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 as follows:

- Comma separated values (`*.csv`)
- Text file with tab delimited
- Microsoft 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()` function 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 file
# location using setwd(). For example setwd("d:/chap2").
# assuming Windows operating system

anscombe <- read.csv("CSVanscombe.csv"",skip=2)
# if the setwd() has not be used then the code will be as
anscombe <- read.csv("d:chap2/CSVanscombe.csv",skip=2)
```

> Note that in the preceding code, `skip=2` argument 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()` function, as shown in the following code:

```
# import csv file that contains both numeric and character variable
# stored in iris.csv file
# 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 when 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()` function, but in this
case 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 had to specify whether the
variable name is present or not, using the argument `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 in a single file. To acquire a dataset for any other type of object 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. See the section, *Modes and classes of R objects* in *Chapter 1*, for more details.

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 function 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")
```

> R can only read Stata 5-12 version data.

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.

Vector and matrix operations

Matrix operation is one of the most commonly used mathematical operations that we perform during data processing and data analysis. All of the matrix operations must be conformable for the operation, mathematically.

The following are the rules that must be followed for matrix operations:

- **Addition or subtraction rule**: There should be at least two vectors, or matrices with the same dimensions
- **Multiplication rule**: There should be at least two vectors or matrices with number of columns of first matrix should be same as the number of rows in second one
- **Element wise multiplication**: For element wise multiplication, both matrices must be of the same dimension

The following is the R code to perform matrix operations:

```
# Creating random matrix with two 3x3 and one 4x3 dimension
# we will use runif() function to generate random number from
# standard uniform distribution
set.seed(1234) # To make the result reproducible
matA <- matrix(rnorm(12),ncol=3)
matB <- matrix(rnorm(9),ncol=3)
matB2 <- matrix(runif(9),ncol=3)

# Matrix addtion addition
matB + matB2# both has dimension 3x3
            [,1]       [,2]       [,3]
[1,] -0.4644296  0.3917120 -0.5932429
[2,]  0.6862780  0.1660850  3.1812950
[3,]  1.2892642 -0.4262042  0.2078681
```

In matrix addition, the default plus (+) symbol works well, but the dimensions of the matrices should be the same. The resultant matrix will also have the same dimensions.

In the following example, we will see if two matrices have different dimensions then matrix addition cannot be performed:

```
 # Matrix addtion addition with varying dimension
matA + matB
Error in matA + matB : non-conformable arrays
```

If the matrices are not of same dimensions, then matrix addition will not work.

```
# Matrix multiplication
matA %*% matB
```

---

[35]

```
          [,1]        [,2]        [,3]
[1,]   0.4230620 0.4281611  1.9715294
[2,] -1.0367218 0.5218026  0.8709481
[3,] -1.3367123 0.6089153 -2.3603264
[4,]   0.8276759 1.4477557  0.5093074
```

In matrix multiplication, the important thing to note is that the symbol is not the default multiplication symbol asterisk (`*`), rather it is `%*%`. If we do not use this symbol, then it will try to perform element wise multiplication. But if the matrix does not have the same dimensions, then the element wise multiplication will not happen, and in that case, an error report will come in.

```
# Multiplication with default multiplication symbol *
matA * matB
Error in matA * matB : non-conformable arrays

# Element wise multiplication
matB * matB2
             [,1]        [,2]         [,3]
[1,] -0.24205483 -0.05536304 -0.204210306
[2,]  0.04008173 -0.34600174  1.849224683
[3,]  0.31641252 -0.44192179  0.009893013

# Matrix multiplication with two 3x3 matrix
# with proper use of symbols %*%
matB %*% matB2
           [,1]        [,2]       [,3]
[1,] -0.5867067 -0.87037213 -0.3355362
[2,]  0.4990147  0.85801532 -0.1971938
[3,] -0.2231869 -0.07027022 -0.4535422
```

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's 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 is 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. Therefore, we should be careful when converting the numeric variable to the factor variable.

Date processing using lubridate

R can handle date variables in several ways. There are built-in R functions available to process 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 work with time zones. Using the POSIXct and POSIXlt class objects, we can work with time zones. But there is another R package, lubridate, contributed by Grolemund and Wickham in 2011, that has a much more user friendly functionality to process date and time, with time zone support. In this section, we will see how we can easily process 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 argument 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 work 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 process date variables in heterogeneous formats. Heterogeneous formats means users can store date information in various ways; for example, the second chapter due on 2013, August, 24, the first chapter submitted on 2013, 08, 18, or 2013 August 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`, it 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 core R function, and also using the `lubridate` package:

```
# Creating date object using base 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
# the output of this section will be vary for the readers
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.

Sometimes we need to change the time zone in date variables for data analysis purposes. For example, we might need to change the time zone from GMT to EST. Using the `_tz()` function in the `lubridate` package made this easy and intuitive to change the time zone. Here is a simple example:

```
date <- ymd("20141221")
date
```

```
[1] "2014-12-21 UTC"
with_tz(date,"EST")
[1] "2014-12-20 19:00:00 EST"
```

Sometimes we need to access individual components of a date and time variable, such as accessing year, month, and day, as well as the days of week, and many more. The following is a list of available easy functions from the `lubridate` packages. These functions are easy to use, and easy to understand.

- To get the year part from a date time variable: `year()`
- To get the month only: `month()`
- To get the week number of a particular date: `weak()`
- To get the day from a date variable (day of month): `day()` or `mday()`
- To get the day number between 1 and 365 (day of year): `yday()`
- To get the day of week: `wday()`
- To get the hour, min, and second: `hour()`, `minute()`, `second()`
- To access the time zone: `tz()`

Here is an example for each of the functions we just listed:

```
date <- ymd("20141221")
year(date)
[1] 2014
month(date)
[1] 12
month(date,label=T)
[1] Dec
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct <
Nov < Dec
month(date,label=T,abbr=F)
[1] December
Levels: January < February < March < April < May < June < July <
August < September < October < November < December
week(date)
[1] 51
day(date)
[1] 21
mday(date)
[1] 21
yday(date)
[1] 355
wday(date)
[1] 1
wday(date,label=T)
```

```
[1] Sun
Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
hour(date)
[1] 0
minute(date)
[1] 0
second(date)
[1] 0
tz(date)
[1] "UTC"
```

Now, we will draw attention to the reader on the output of `hour()`, `minute()`, and `second()`; the output of these functions is `zero`, which means that the date object contains only the date part, and as a result, the time part is set to `zero`. So, the results indicate that the date is recorded at 12:00 AM. At the point we change the time zone of the date object, the value will be different; here is an example:

```
hour(with_tz(date,"EST"))
[1] 19
```

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:
 	
`nchar(c("x","y",NA))`
`[1] 1 1 2`
	`str_length()`: This is the same as `nchar()`, but it preserves NA. For example:
	
	`str_length(c("x","y",NA))`
	`[1] 1 1 NA`

Base R functions	stringr functions
`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 a 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))

data_2variable
   x1 x2
1  2  5
2  3  6
3  4  7
4  5  8
5  6  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, the index `[[1]]` gets 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.
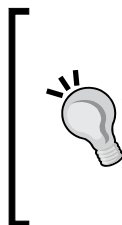
Downloading the example code

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.

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 and a state-of-the-art approach in the group-wise data manipulation using R.

3
Data Manipulation Using plyr and dplyr

We often collect data across different places and time points and across human characteristics. A census collects data across different states. In a longitudinal study, we collect information over different time points. Those individuals could be male or female, and their occupation could be different. All individuals under any study could be split into different groups based on these geographical, temporal, and occupational characteristics. We usually analyze data as a whole, but sometimes it is useful to perform some tasks separately among different groups.

As an example, if we collect details of the income of different individuals from six different regions, then we might be interested in seeing the income distribution among different professions (considering five different professions), across six regions. This income could vary depending on whether the person is a male or female. In this situation, we can conceptualize this problem by splitting the dataset based on profession, gender, and region. There should be `5 x 6 x 2=60` different groups, and we need to calculate the average income separately for each groups. Finally, we want to combine the result to see all the information side by side. This group-wise operation is often termed as the split-apply-combine approach of data analysis.

In this approach, first we split the dataset into some mutually exclusive groups. We then apply a task on each group and combine all the results to get the desired output. This group-wise task could be generating new variables, summarizing existing variables, or even performing regression analysis on each group. Finally, combining approaches helps us get a nice output to compare the results from different groups.

This chapter will deal with the following topics:

- Applying the split-apply-combine strategy
- Utilities of the `plyr` library
- Different functions in the `plyr` package for handling different data structures
- Comparing base R and `plyr`
- Powerful data manipulation with the `dplyr` library

Applying the split-apply-combine strategy

For the purpose of demonstration, we will use an iris flower dataset, which is readily available in R. The iris flower has three different species: iris setosa, iris virginica, and iris versicolor. Fifty samples from each species were collected and, for each sample, four variables were measured: the length and width of the sepals and petals. The name of each flower is stored under the species column, and the length and width of sepal is stored under the `Sepal.Length` and `Sepal.Width` columns, respectively. Similarly, the length and width of the petal are stored under the `Petal.Length` and `Petal.Width` columns, respectively. The following command shows the first few rows from the iris data frame:

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Now we will use the split-apply-combine strategy to find the average width and length of sepal and petal for three different species of iris. The strategy will be as follows:

1. First we will split the dataset into three subsets according to the species of the flower.
2. Next, for each subset, we will compute the average width and length of the sepal and petal.

3. Finally, we will combine all the results to compare them with each other.

```
# Step 1: Splitting dataset
iris.setosa <- subset(iris,Species=="setosa",
select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))

iris.versicolor <- subset(iris,Species=="versicolor",
select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))

iris.virginica <-  subset(iris,Species=="virginica",
select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))

# Step 2: Applying mean function to calculate mean
setosa <- colMeans(iris.setosa)
versicolor <- colMeans(iris.versicolor)
virginica <-colMeans(iris.virginica)

# Step 3: Combining results
rbind(setosa=setosa,versicolor=versicolor,virginica=virginica)
```

This is the detailed code to implement the split-apply-combine approach. We could implement the strategy with less code, as follows:

```
# Step 1: Splitting dataset
iris.split <- split(iris,as.factor(iris$Species))

# Step 2: Applying mean function to calculate mean
iris.apply <- lapply(iris.split,function(x)colMeans(x[-5]))

# Step 3: Combining results
iris.combine <- do.call(rbind,iris.apply)
iris.combine
           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
```

In later sections in this chapter, we will see how the `plyr` package comes in handy for implementing the split-apply-combine approach on all kind of data structures. Using the `plyr` package, one line of code would be sufficient to implement these three steps.

Introducing the plyr and dplyr libraries

We have seen how we can implement the split-apply-combine approach on a data frame using three lines of code. The `plyr` package helps us to implement the approach in one line. Since R has multiple data structures, we need multiple functions to work on different data structures. R has three main data structures: list, array, and data frames. So, there could be three different types of input, and the output could produce three different types of data structures. There could be `3 x 3 = 9` possible input-output combinations, and for this reason, `plyr` has `9` functions to incorporate all the input-output combinations. In addition, we have three additional functions that take six different types of input but display only one type of output.

The `plyr` package works on every type of data structure, whereas the `dplyr` package is designed to work only on data frames. The `dplyr` package offers a complete set of functions to perform every kind of data manipulation we need in the process of analysis. These functions take a data frame as the input and also produce a data frame as output; hence the name: `dplyr`. There are two different types of function in the `dplyr` package: a single-table function and an aggregate function. The single-table function takes a data frame as input and takes an action, such as subsetting the data frame, generating new columns in the data frame, or rearranging the data frame. The aggregate function takes a column as input, and produces a single value as output, which is mostly used for summarizing columns. These functions do not allow us to perform any group-wise operation, but let's combine these functions with the `group_by()` function. This allows us to implement the split-apply-combine approach.

plyr's utilities

The most important utility of the `plyr` package is that a single line of code can perform all the split, apply, and combine steps. What we have done using three lines of code in the first section can be implemented in just one line using the `plyr` package:

```
library(plyr)
ddply(iris, .(Species), function(x) colMeans(x[-5]))
```

Here, `ddply()` is a function from the `plyr` package, which takes a data frame as input and produces a data frame as output. Hence, the name of the function is `ddply`. Here, the argument works as follows:

- The first argument is the name of the data frame. We put `iris`, since the iris dataset is in the data frame structure and we want to work on it.

- The second argument is for a variable or variables, according to which we want to split our data frame. In this case, we have `Species`.

- The third argument is a function that defines what kind of task we want to perform on each subset.

One question that should come into our mind is how the function takes data as input. Here, we will split the data frame into three different groups as follows:

- The first subset, which is also a data frame, will be considered as an input of the function. The function will calculate all the column means and store them somewhere.
- The second subset will be considered as input for the function, and so on.
- All the outputs will be combined to form a single data frame.

This is like the `bysort` command in `Stata`, but with a lot more flexibility. Since there are different types of data structure in R, one single function cannot handle all types of data structure. That is why we have multiple functions in the `plyr` package that have a very similar naming convention. It is very easy to remember all the functions, and it is easy to apply them when we need.

Intuitive function names in the plyr library

To perform any kind of data processing, we need to know the type of input that we have to provide and the expected output format. In most R functions, it is difficult to understand from function names what types of input they accept and what the expected types of output are. Function names in the `plyr` package are much more intuitive and instructive about their input and output types, compared to any other available packages. Each function is named according to the type of input it takes, 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 an 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 12 functions. Instead, 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 involving 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 `*ply()`, it indicates that the 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 `iris3 dataset`; this is the same data, but it is 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
```

Since `iris3` is an array, we need to specify according to which dimension we will split the array. We specify this using the `.margins` parameter, in the `adply` function. We put `.margins=3` in adply function as: `adply(iris3,.margins=3,colMeans)` to tell the `adply` function that we want the splitting according to the third dimension of a three dimensional array object. If we wanted to split the data according to row or column, we would put `1` or `2`, respectively. It is also legitimate to use a combination of dimensions. In that case, `c(1,2)` could be a choice.

The following code snippet calculates the column mean for each species, with the input as an array as well as the output as arrays:

```
# 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
```

Inputs 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 inputs and arguments. The rules of splitting are described shortly in this section.

Arrays are sliced by dimension into lower dimensional pieces. The corresponding common function is `a*ply()`, where the array is the common input, and the output can be an array, data frame, or list.

Data frames are sliced and subset by a combination of variables from the input 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 being split, 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 into 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 manner similar to the `apply()` function in base R. It does the following:

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

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 a 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:

Sample size (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
```

Comparing base R and plyr

In this section, we will compare the 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 five-number summary statistics, 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)
}
```

Here, you can see 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
```

Powerful data manipulation with dplyr

Mostly, in real-life situations, we usually start our analysis with a data frame-type structure. What do we do after getting a dataset and what are the basic data-manipulation tasks we usually perform before starting modeling? They are explained here:

1. We check the validity of a dataset based on conditions.

2. We sort the dataset based on some variables, in ascending or descending order.

3. We create new variables based on existing variables.

4. Finally, we summarize them.

This is a list of tasks we usually perform over full datasets. The `dplyr` package has all the necessary functions to perform all the tasks listed and some more additional tasks that come in handy in the data-manipulation process. Group-wise operation is also possible using the `dplyr` package. In the `dplyr` package, every task is performed using a function that is called a verb. We may need to use multiple verbs on the same data frame. This could force us to write either a very long line or multiple lines of code. Chaining is a powerful feature of `dplyr` that allows the output from one verb to be piped into the input of another verb using a short, easy-to-read syntax.

Filtering and slicing rows

Sometimes, it is more important to subset the data frame based on values of a variable or multiple variables. The `filter()` function allow us to perform this task. If we want to just see all the observations under the virginica species, then we need to use the following code:

```
filter(iris,Species=="virginica")
```

We could also create a data frame with sepal length less than 6 cm and sepal width less than or equal to 2.7 cm:

```
filter(iris,Species=="virginica" &  Sepal.Length<6  & Sepal.
Width<=2.7)
```

We could also extract the subset of a data frame using the `slice()` function. If we want to subset the first 10 observations, the last 10 observations, or even the 95th to 105th observation, then we could use the following code, respectively:

```
slice(iris, 1:10)
slice(iris, 140:150)
slice(iris, 95:105)
```

Arranging rows

To sort the whole data frame based on a single variable or multiple variables, we could use the `arrange()` function. We could sort the dataset according to the lowest length of sepal to the highest length of sepal:

```
arrange(iris, Sepal.Length)
```

We could also sort the dataset by sorting the data frame for sepal length and then for sepal width:

```
arrange(iris, Sepal.Length, Sepal.Width)
```

If we want to sort the data frame in ascending order for sepal length, but descending order for sepal width, we can use the `desc()` function from this package:

```
arrange(iris, Sepal.Length, desc(Sepal.Width))
```

It seems that the `arrange()` function in the `dplyr` package is very similar to the `order()` function, but it has a lot more flexibility and an intuitive structure of input arguments.

Selecting and renaming

Most of the time, we do not work on all the variables in a data frame. Selecting a few columns could make the analysis process less complicated. We could easily select a smaller number of columns from a data frame. In our example, we selected the `Sepal.Length` and `Sepal.Width` species of the iris data frame using the `select()` function:

```
select(iris, Species, Sepal.Length, Sepal.Width)
```

We could also change the column name using the `rename()` function:

```
rename(iris, SL=Sepal.Length, SW= Sepal.Width, PL=Petal.Length, PW=
Petal.Width )
```

Adding new columns

Very often, we need to create new columns for the purpose of analysis. In the iris data frame, if we want to convert the width and length of sepal and petal from centimeter to meter, we could use the `mutate()` function as follows:

```
mutate(iris, SLm=Sepal.Length/100, SWm= Sepal.Width/100, PLm=Petal.
Length/100, PWm= Petal.Width/100 )
```

Also, we could standardize these variables in the following way:

```
mutate(iris, SLsd=(Sepal.Length-mean(Sepal.Length))/sd(Sepal.Length),
                    SWsd= (Sepal.Width-mean(Sepal.Width))/sd(Sepal.
Width),
                    PLsd=(Petal.Length-mean(Petal.Length))/sd(Petal.
Length),
                    PWsd= (Petal.Width-mean(Petal.Width))/sd(Petal.
Width) )
```

If we want to keep only the new variables and drop the old ones, we could easily use the `transmute()` function:

```
transmute(iris, SLsd=(Sepal.Length-mean(Sepal.Length))/sd(Sepal.
Length),
                    SWsd= (Sepal.Width-mean(Sepal.Width))/sd(Sepal.
Width),
                    PLsd=(Petal.Length-mean(Petal.Length))/sd(Petal.
Length),
                    PWsd= (Petal.Width-mean(Petal.Width))/sd(Petal.
Width) )
```

Selecting distinct rows

We can extract distinct values of a variable or multiple variables using the `distinct()` function. Sometimes, we might encounter duplicate observations in a data frame. The `distinct()` function helps eliminates these observations:

```
distinct(iris,Species,Petal.Width)
```

Column-wise descriptive statistics

We could summarize different variables based on different summary statistics using the `summarise()` function. Here, we summarized the length and width of sepal and petal by calculating their average:

```
summarise(iris, meanSL=mean(Sepal.Length),
                            meanSW=mean(Sepal.Width),
                            meanPL=mean(Petal.Length),
                            meanPW=mean(Petal.Width))
```

Group-wise operations

The functions we discussed in previous sections from the `dplyr` package work on the whole data frame. If we want to use a group-wise operation on different columns, we need to use a combination of the `group_by()` function and the other functions:

```
iris.grouped<- group_by(iris, Species)
summarize(iris.grouped, count=n(),
                  meanSL= mean(Sepal.Length),
               meanSW=mean(Sepal.Width),
               meanPL=mean(Petal.Length),
               meanPW=mean(Petal.Width))
```

Here, the combination of the `group_by()` and `summarise()` functions could be considered as an implementation of the split-apply-combine approach on a data frame. Here, `group_by()` takes the data frame as an input and produces a data frame too. However, this data frame is a special type of data frame where grouping information is stored inside it. When this special type of data frame is supplied as an input of the `summarise()` function, it knows that the calculation should be group-wise. Here, all the calculations using `n()`, `mean()` are performed group-wise.

Chaining

Sometimes, it could be necessary to use multiple functions to perform a single task. From the iris data, we may want to use the `group_by()` operation to get a special data frame. Then we may want to use the `select()` function to select only the sepal length and width. It would then be interesting to see location and dispersion summary statistics. Finally, we might want to see species with maximum average sepal length and maximum average sepal width:

```
iris
iris.grouped<- group_by(iris, Species)
iris.grouped.selected<- select(iris.grouped, Sepal.Length, Sepal.
Width)
iris.grouped.selected.summarised<- summarise(iris.grouped.selected,
                  meanSL=mean(Sepal.Length),
                  sdSL=sd(Sepal.Length),
                meanSW= mean(Sepal.Width),
                sdSW= sd(Sepal.Width))
filter(iris.grouped.selected.summarised, meanSL==max(meanSL) |
meanSW==max(meanSW))
```

The workflow is very intuitive but, each time we applied a function, we saved a new data frame. The `dplyr` package has a nice operator that prevents us from saving a new data frame each time we perform an action on it. This operator is called the `%>%` chain operator; it is similar to the pipe operation in shell scripting. The `%>%` operator turns `x %>% f(y)` into `f(x,y)`. This operator not only allow us to save storage, but also makes the code cluster more intuitive for other people to understand, It also helps you read your code in future:

```
iris %>%
      group_by( Species) %>%
      select(Sepal.Length, Sepal.Width)   %>%
      summarise( meanSL=mean(Sepal.Length),
                          sdSL=sd(Sepal.Length),
                          meanSW= mean(Sepal.Width),
                          sdSW= sd(Sepal.Width)) %>%
      filter(meanSL==max(meanSL) | meanSW==max(meanSW))
```

When we have a script file with a huge number of lines, this feature comes in handy. A cluster of these lines of code in a script file will help us understand that these lines of code were written to perform one task. This also saves us the effort of writing an additional data frame name each time.

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 it 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 types. A large variety of data processing can be done using only a few functions with common input and various types of output. For further reading, an interested user can refer to the paper *The Split-Apply-Combine Strategy for Data Analysis* by Wickham, which can be found at `http://www.jstatsoft.org/v40/i01/paper`. We also discussed how we can use `dplyr` as a powerful tool to manipulate data frame.

In the following chapter, you will learn about reshaping a dataset, which is another important aspect of group-wise data manipulation.

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 a reorientation to perform certain types of analysis. The layout of datasets could be long or wide. In a long layout, multiple rows represent a single subject's record, whereas, in a wide layout, a single row represents a single subject's record. Statistical analysis sometimes requires wide data and sometimes long data. 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 paradigms of reshaping datasets, melt and cast 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 (refer to Reshaping Data with the reshape Package paper by Hadley 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 understand how we can change the layout using the new paradigm of reshaping datasets with `melt` and `cast`. To run the example of this chapter, you need to install both the `reshape` and `reshape2` packages.

Typical layout of a dataset

A single dataset can be rearranged in many different ways but, before going into this rearrangement, let's look 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 the role of 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. They are explained here:

- **Identifier variables**: These help 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 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 a wide layout and a 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 the 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 a long layout to a wide one and vice versa using R.

New layout of a dataset

In R, the layout of a dataset is known to be different from the typical layout that we discussed in the previous section. This new layout 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 and 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 a 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. This 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 indicate 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` repeat for each subject.

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

```
# 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 should use the `reshape` package instead.

Reshaping the dataset with the reshape package

As we have seen, there are two different paradigms to define the layout of a dataset. To change the layout of a dataset, here are the steps of a new paradigm. We need to use the `reshape` package, 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, *New layout of a dataset*, we talked about 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 to melt a data frame. To perform melting operations using the melt function, we need to know what the `identification` variables and `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 that 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. This gives the flexibility to deal with 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 value in practice: sampling zero (that is, no response) and structural missing. The sampling zero values are explicitly coded and represented in the dataset, but the structural missing values depend on the structure of the dataset. Structural missing value are implicit in the dataset; they are 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 is taken from the Reshaping Data with the reshape Package paper by Hadley Wickham, which can be found at `http://www.jstatsoft.org/v21/i12/paper`. It clearly explains implicit and explicit missing values 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 value. 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. They are as follows:

- `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 to perform 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 will now rearrange the data in such 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 will rearrange the data again in such 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 that the column names of the last two examples are not valid column names because they contain numbers. This 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, there are 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

The `melt` function in the `reshape2` package works the same as the `melt` function in the `reshape` package. The only difference is that the `melt` function in the `reshape2` package is faster and more memory-efficient than the `melt` function in the `reshape` package. The `melt` function is pretty efficient at converting all data structures to molten data frames. The next step is to reshape the molten data frame into either a data frame or array structure. In the `reshape` package, this task is done using only the `cast` function. The output of the `cast` function, whether a data frame or array, depends on how we put the formula. In the `reshape2` package, we have the `dcast` function to produce the data frame as output and `acast` to produce an array from a molten data frame.

We will also use the `students` dataset here. First, we will melt the dataset using the `melt` function in the `reshape2` package, and then we will illustrate how we can use the `dcast` and `acast` functions to reshape the data:

```
library(reshape2)
molten_students <- melt(students,id.vars=c("sid","exmterm"))
```

The basic casting formula is `x_variable + x_2 ~ y_variable + y_2 ~ z_variable ~`. For the purpose of illustration, consider `x_variable, x_2` as the first set of variables, `y_variable, y_2` as the second set of variables, `z_variable, z_2` as the third set of variables, and so on. The first set of variables is used to make the row uniquely identifiable. For the molten dataset `molten_students` we are considering `sid` as first set of variable and variable as second set of variable in the following example:

```
> dcast(molten_students, sid~variable)
Aggregation function missing: defaulting to length
  sid math literature language
1   1    2          2        2
2   2    2          2        2
```

Here, we can see that we have only two rows, although we do not have all the data here. This happened because the `sid` variable has only two unique values. To make the column uniquely identifiable using just the `sid` variable, we only need two rows:

```
> dcast(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
```

Now we have four rows, because the `sid` and `exmterm` variables together can create only four unique rows. We have complete data here. So, in the process of data analysis, we should use the entire identification variable as the first set of variables. This is also true for the `acast` function:

```
> acast(molten_students, sid~variable)
Aggregation function missing: defaulting to length
  math literature language
1    2         2        2
2    2         2        2
```

Here, there is no `sid` variable in the data, because `acast` produces an array and the value of the `sid` variable is used as the row index for this data:

```
> acast(molten_students, sid+exmterm~variable)
    math literature language
1_1   50         40       70
1_2   65         45       80
2_1   75         55       75
2_2   69         59       78
```

This sheds light on how the combination of the `sid` and `exmterm` variables is considered as an index of the output array.

The second set of variables is used to produce column name. The combination of the values of the second set of variables is used as the column name of the output data frame in the `dcast` and `acast` functions:

```
> dcast(molten_students, sid~variable+exmterm)
  sid math_1 math_2 literature_1 literature_2 language_1 language_2
1   1     50     65           40           45         70         80
2   2     75     69           55           59         75         78

> acast(molten_students, sid~variable+exmterm)
  math_1 math_2 literature_1 literature_2 language_1 language_2
1     50     65           40           45         70         80
2     75     69           55           59         75         78
```

Here, we can see that the combination of the second set of variables is considered as the column name of the output data frame and array.

The third set of variables is only applicable for the `acast` function since an array could go beyond two dimensions, but data frame is strictly restricted to two dimensions. This is why we could not use the third set of variables in the formula for the `dcast` function:

```
> acast(molten_students, sid~exmterm~variable)
, , math

    1  2
1 50 65
2 75 69


, , literature

    1  2
1 40 45
2 55 59


, , language

    1  2
1 70 80
2 75 78
```

Summary

This chapter introduced a 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, employing only two functions allows users to rearrange datasets into various layouts as required. This chapter also discussed structural missing, sampling zero values, and how to deal with these missing values during the melting process. For faster and large data rearrangement, you 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.

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 2^{31}-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
```

The topic of database administration is beyond the scope of this book, but we can easily discuss connectivity with databases using R.

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. The process of creating a connection is beyond the scope of this book, but we will learn about the topic briefly.

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, 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 the large dataset, `large.dat`, 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 $2^{31}-1$ 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_Wid
th) from iris group by Species")

colnames(iris_avg) <- c("Species","Sepal_L","Sepal_W","Petal_L","Peta
l_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.

6
Text Manipulation

Text data is one of the most important areas in the field of data analytics. Every day, we are producing a huge amount of text data through various media. For example, Twitter posts, blog writing, and Facebook posts are major sources of text data. Text data can be used to retrieve information in sentiment analysis and even entity recognition. In this chapter, we will discuss how R can be used to process text data, which we can utilize in any text analytics areas. These types of data can also be used in text categorization, predictive analytics, lexical analysis, document summarization, and even in natural language processing. First, we will discuss the default functions of R for processing text data. Then, we will introduce a `stringr` library to work with text data. We will cover the following topics in this chapter:

- What is text data?
- Sources of text data
- Obtaining text data
- Text processing using default functions
- Text processing using `stringr`
- Structuring text data for text mining

Text data and its source

Text data is any type of text on any topic. Here is a list of text data and its sources:

- Tweets from any individual, or from any company
- Facebook status updates
- RSS feeds from any news site
- Blog articles

---

[95]

- Journal articles
- Newspapers
- Verbatim transcripts of an in-depth interview

These are the most common sources of text data. In the area of text analytics, Twitter data has been used frequently to find topic trends through topic modeling. Text data has also been used to predict certain diseases from tweets. The HTML web file are also a great source of text data.

Getting text data

Text data can be embedded into any dataset as a string variable. Also, text data can be stored as plain text files even in the HTML file format. In this section, we will see how we can read or import text data into the R environment for further processing.

The easiest way to get text data is to import from a `.csv` file where some of the variables contain character data. For example, the `tweets.csv` file contains 50 Twitter statuses on a certain topic. Since this is a `.csv` file, we can import it using the `read.csv()` function, but we have to protect automatic factor conversion by specifying the `stringsAsFactors=FALSE` argument. An example of importing text data from the `tweets.csv` file is as follows:

```
textData <- read.csv("tweets.csv",stringsAsFactors=FALSE)
str(textData)

''data.frame'':   50 obs. of  2 variables:
 $ ID    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ TWEETS: chr  "Sohum Spa at Movenpick HotelSpa Bangalore reveals
Indian traditions for relaxation" "Sohum Spa at Movenpick HotelSpa
Bangalore reveals Indian traditions for relaxation" "SalesMarketing
Manager at Prestige Leisure Resorts Pvt Ltd" "Assistant Front Office
Manager at The LEELA PalaceBangalore" ...
```

So, this is just as simple as importing any other data in R. Now, let's look at an example of obtaining text data from a plain text file. The `tweets.txt` file is the plain text file. We will import this file using the generic `readLines()` function:

```
textData1<-readLines("tweets.txt")"")
str(textData1)
 chr [1:51] "ID\tTWEETS " "1\tSohum Spa at Movenpick HotelSpa
Bangalore reveals Indian traditions for relaxation" ...
```

If we see the structure of the `textData` and `textData1` objects, there is a difference. The `textData` object, which is imported by `read.csv()`, is a data frame, whereas the `textData1` object imported by `readLines()` is just a vector of characters. To convert the character vector into a data frame, we need some basic processing. We will talk about this in a later section. Importing text data from an HTML page, which is technically known as web scraping, is one of the widely used sources of text data. In this example, we will see how we can import data from the web. Interestingly, the `readLines()` generic function can be used to read an HTML file too, but later on, we need to process it to have a structured database. Here is an example to importing a Wikipedia article:

```
# Creating object with the URL
conURL <- "http://en.wikipedia.org/wiki/R_%28programming_language%29"

# Establish the connection with the URL
link2URL <- url(conURL)

# Reading html code
htmlCode <- readLines(link2URL)

# Closing the connection
close(link2URL)

# Printing the result
htmlCode
```

Like the previous `textData1` object. This is also a character string, but this time it contains the HTML code. From this HTML code, we are able to generate structured data for further use.

To obtain text data from social networking sites such as Twitter and Facebook, there are designated R libraries. To extract Twitter data, we can use `tweetR` and, to extract data from Facebook, we could use `facebookR`. In the section *Working with Twitter data*, we will discuss this in detail.

The `tm` text mining library has some other functions to import text data from various files such as PDF files, plain text files, and even from doc files. Readers are advised to look into the `tm` library for further information. Discussing the `tm` library in detail is beyond the scope of this chapter.

Text processing using default functions

Some of you might not be interested in text mining, but you still need to process text data in your day-to-day activities. In this section, we will try to give some examples that will be helpful for your daily needs. The following are the general tasks that we need to perform frequently:

- Removing certain characters or words from a string
- Splitting the character string to get structured information
- Matching certain parts of the characters to find out some patterns
- Changing lowercase to uppercase, and vice versa
- Calculating the number of characters in a string
- Extracting a certain part from a string
- Extracting only digits from a string

We will see an example for each case listed previously. First, we will remove a certain word from a string. To do so, we will use the textData object. This object has two variables, and one of them contains text data. We will use the first observation from that text variable:

```
# Extracting first observation
text2process <- textData$TWEET[1]
text2process
[1] "Sohum Spa at Movenpick HotelSpa Bangalore reveals Indian
traditions for relaxation"
```

Now, we are interested in removing the prepositions, such as *for* and *at*, from the text. To do so, we will use the gsub function, which replaces certain text based on pattern matching. The important arguments of the gsub function are pattern, replacement, and the string is as follows:

```
prepRemovedText <- gsub(pattern="for",replacement="",x=text2process)
prepRemovedText
[1] "Sohum Spa at Movenpick HotelSpa Bangalore reveals Indian
traditions  relaxation"
```

This example shows that the word *for* has been removed from the original string.

We can also split the string so that it has a different data structure. For example, if we split the text2process object using the splitting character as a blank space, then it will be a vector of the character, with each word separated. Here is an example:

```
splittedText <- strsplit(text2process,split=" ")
splittedText
```

```
[[1]]
 [1] "Sohum"     "Spa"        "at"          "Movenpick" "HotelSpa"
"Bangalore" "reveals"    "Indian"      "traditions" "for"
"relaxation"
```

The `strsplit` function takes a character string as input and the character in `split` argument specifies the location of split. The output is initially stored in a list object, but to get the output as a vector, we can remove the object from the list in the following way:

```
unlist(splittedText)
 [1] "Sohum"     "Spa"        "at"          "Movenpick" "HotelSpa"
"Bangalore" "reveals"    "Indian"      "traditions" "for"
"relaxation"
```

Converting `lowercase` and `uppercase` strings is another important function when we work with text data. Since R is case-sensitive, the words `Spa` and `spa` are different, though, in fact, they are the same word. So, to remove ambiguity, we can convert either all the words to lowercase or change them all to uppercase.

To convert into lowercase and then to uppercase, let's take a look at the following example:

```
tolower(text2process)
[1] "sohum spa at movenpick hotelspa bangalore reveals indian
traditions for relaxation"

toupper(text2process)
[1] "SOHUM SPA AT MOVENPICK HOTELSPA BANGALORE REVEALS INDIAN
TRADITIONS FOR RELAXATION"
```

During data analysis and in text processing, we need to know the number of characters in a character string. For example, in some databases, the `id` variable could be text, and it should contain a certain number of characters. In this case, we need to count whether the required number of characters is present or not. In this example, we will see how to calculate the number of characters from a string. The total number of characters in the `text2process` string can be found using the `nchar()` function. This function counts each character, including a blank space:

```
nchar(text2process)
[1] 82
```

Now, we will pass the same function, but this time the input will be the unlisted split character vector:

```
nchar(unlist(splittedText))
 [1]  5  3  2  9  8  9  7  6 10  3 10
```

This time, we have a vector of an integer because the input of the `nchar()` function, here, is a vector of the character object. So, it returns the number of characters for each component of that input vector.

In some cases, the text variable contains both date and time information. For example, `02Feb2015:11:15PM` is a character string. We need to extract only the date part for further processing. To do the task, take a look at this example:

```
# Creating the character string with date and time information
dateTimeobject <- "02Feb2015:11:15PM"

# Extracting only the character between 1 to 9
# including 1st and 9th
substr(dateTimeobject,1,9)
[1] "02Feb2015"
```

So, the `substr` function can be used to extract a portion of text from a character string.

During text processing, sometimes, we need to extract only the digits from a character string. In the example, we will see how we can do this task. In R, we have default color names that can be accessed through the `color()` function. Some of the color names contains digits such as `red1`, `red2`, and so on. In this example, we will extract only the digits from color names:

```
# to see the color names
colors()
# Now to extract the digit from the color names
as.integer(gsub("\\D", "", colors()))
```

This table gives us an idea about the facilities in the `stringr` library and its link with the default R functions:

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. Here is an example:
 	
`nchar(c("x","y",NA))`
`[1] 1 1 2`
	`str_length()`: This is the same as `nchar()`, but it preserves `NA`. Here is an example:
	
	`str_length(c("x","y",NA))`
	`[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-hand side 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 white spaces.
Unavailable	`str_pad()`: This is used to pad a string with extra white spaces on the left-hand side, right-hand side, or both sides.

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

Working with Twitter data

Twitter is one of the best sources of text data. In this section, we will extract text data from Twitter using the `#rstats` hashtag. After extracting the text, we will clean it and then produce a wordcloud. The required libraries for this particular section are as follows:

- `twitteR`
- `tm`
- `wordcloud`

To extract data from Twitter, first, we need to connect with the Twitter account through a valid authentication process. The code to authenticate the R session with Twitter to extract data, is as follows:

```
library(twitteR)
# need to provide actual string for each key by replacing xxxx
setup_twitter_oauth(consumer_key="xxxx",
                    consumer_secret="xxxx",
                    access_token="xxxx",    access_secret="xxxx")
```

Once the authentication process is complete, we can extract text data. In the following code, we extracted data for 500 tweets with a `#rstats` hashtag. We restricted the tweets to English only. The results of this section will be changed over time.

```
# Extracting 500 recent tweets with #rstats hashtag
# and language of the tweets is English
tweets<-searchTwitter("#rstats", n=500,lang='en')
```

The object will be listed in nature with lots of information related to tweets. We can easily check the structure of the newly created objects using the following `str()` function:

```
str(tweets)
```

To prepare a word cloud, we will use only the status text from the extracted tweets. To do so, we need to convert the list object into a data frame and then take only the text column. The code chunk to get only the text column is as follows:

```
datTweet<-plyr::ldply(tweets,as.data.frame)
vecStatus <- datTweet$text
```

The next step is to clean the text by removing HTML tags, retweets tags (RT), and punctuation symbols:

```
#Clean Text
vecStatus = gsub("(RT|via)((?:\\b\\W*@\\w+)+)","",vecStatus)
vecStatus = gsub("http[^[:blank:]]+", "", vecStatus)
vecStatus = gsub("@\\w+", "", vecStatus)
vecStatus = gsub("[ \t]{2,}", "", vecStatus)
vecStatus = gsub("^\\s+|\\s+$", "", vecStatus)
vecStatus <- gsub('\\d+', '', vecStatus)
vecStatus = gsub("[[:punct:]]", " ", vecStatus)

# additional cleaning by tm library
library(tm)
corpus = Corpus(VectorSource(vecStatus))
corpus = tm_map(corpus,removePunctuation)
corpus = tm_map(corpus,stripWhitespace)
corpus = tm_map(corpus,tolower)
corpus = tm_map(corpus,removeWords,stopwords("english"))
```

Once we clean the text, it is now ready to produce a word cloud of the texts. To make a word cloud, we will call the `wordclund` library and then use the `wordcloud()` function, as shown in the following example:

```
library(wordcloud)
wordcloud(corpus)
```

The following screenshot shows the wordcloud corresponding to the 500 extracted tweets using rstats hashtag:

Summary
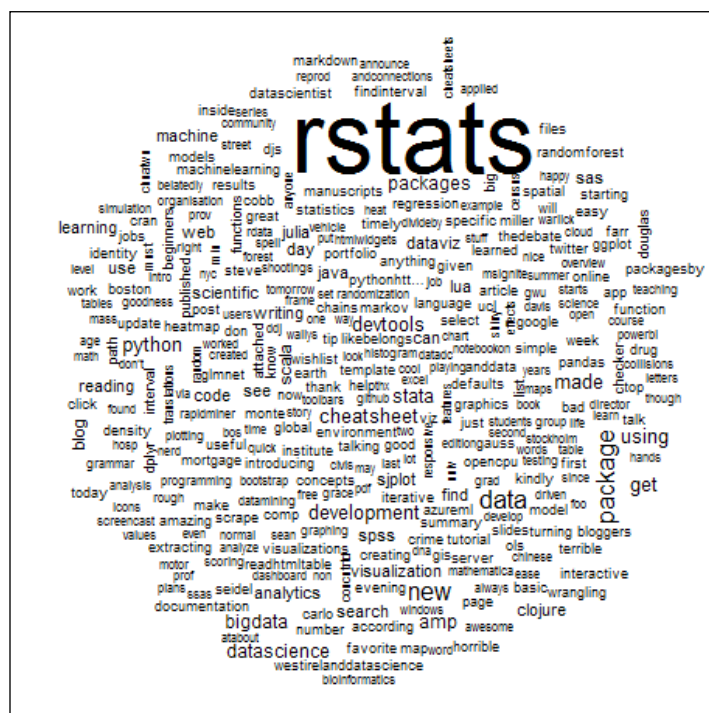
In this chapter, we tried to discuss the source of text data and how plain text can be handled using R. We also compared functions from the `stringr` library with the default R functions to process text data. In the final section, we showed you how to extract text data from Twitter posts and then clean the data to produce a word cloud. The processed text data can be used in other text-mining applications, such as topic modeling and sentiment analysis.

Index

A

array 36

B

Base R functions
  about 54
  nchar() 54
  paste() 54
  substr() 55

C

character manipulation 54, 55
columns, dplyr
  adding 73, 74
column-wise descriptive statistics 74
**Comprehensive R Archive Network
        (CRAN)**
  URL 12
Coordinated Universal Time (UTC) 51

D

data
  acquiring 41-44
  manipulating, sqldf package used 100-102
  melting 83, 84
data frame 31-33
data reshaping 77
dataset
  about 77
  layout 78-81
  reshaping, from typical layout 81, 82
  reshaping, reshape package used 82

dataset layout
  identifier variables 78
  long layout 78
  measured variables 78
  wide layout 79
default functions
  used, for text processing 108-111
dplyr
  about 62
  chaining 75, 76
  columns, renaming 73
  columns, selecting 73
  column-wise descriptive statistics 74
  data manipulation 72
  distinct rows, selecting 74
  group-wise operations 75
  new columns, adding 73, 74
  rows, arranging 73
  rows, filtering 72
  rows, slicing 72
dumpObjects utility 96
dumpImage utility 96

E

Excel file
  about 93
  ODBC connection string, creating 93

F

factor
  about 29, 30
  manipulation 46-48

S

split-apply-combine strategy
  applying 60, 61
sqldf package
  tasks, performing 99
  used, for data manipulation 100-102
stringr functions
  about 54
  str_c() 54
  str_dup() 55
  str_length() 54
  str_pad() 55
  str_sub() 55
  str_trim() 55
subscripting 55-57
subsetting 55-57

T

text
  processing, default functions used 109-111
  data 105, 106
  data, getting 106, 107
Twitter data
  working with 111-113

V

vector
  about 27-29
  operation 45, 46

W

Wickham
  URL 111

Thank you for buying
Data Manipulation with R
Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
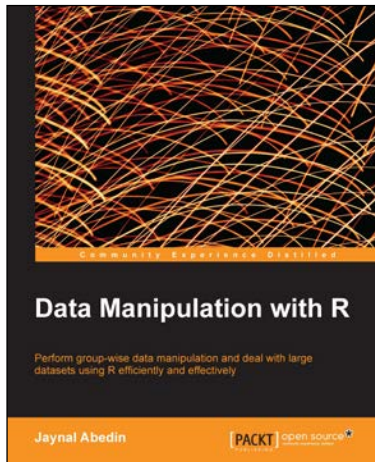
Data Manipulation with R

ISBN: 978-1-78328-109-1   Paperback: 102 pages

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

1. Perform factor manipulation and string processing.

2. Learn group-wise data manipulation using plyr.

3. Handle large datasets, interact with database software, and manipulate data using sqldf.
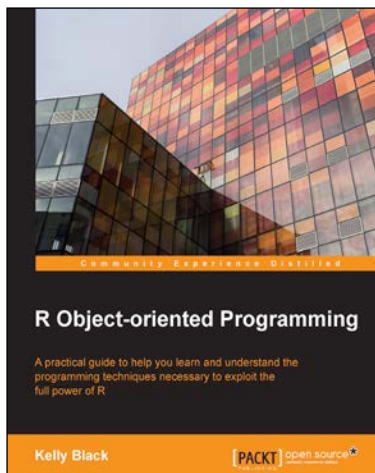
R Object-oriented Programming

ISBN: 978-1-78398-668-2   Paperback: 190 pages

A practical guide to help you learn and understand the programming techniques necessary to exploit the full power of R

1. Learn and understand the programming techniques necessary to solve specific problems and speed up development processes for statistical models and applications.

2. Explore the fundamentals of building objects and how they program individual aspects of larger data designs.

3. Step-by-step guide to understand how OOP can be applied to application and data models within R.

Please check **www.PacktPub.com** for information on our titles
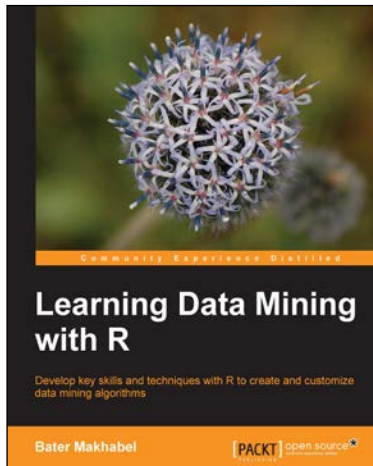
Learning Data Mining with R

ISBN: 978-1-78398-210-3   Paperback: 314 pages

Develop key skills and techniques with R to create and customize data mining algorithms

1. Develop a sound strategy for solving predictive modeling problems using the most popular data mining algorithms.

2. Gain understanding of the major methods of predictive modeling.

3. Packed with practical advice and tips to help you get to grips with data mining.
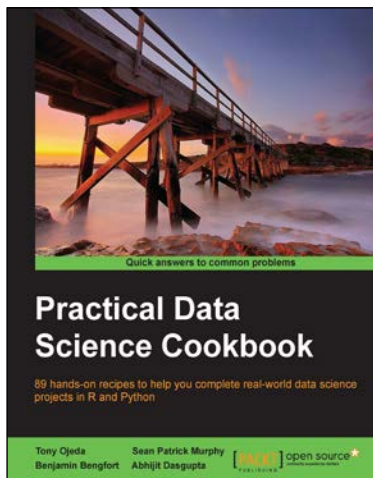
Practical Data Science Cookbook

ISBN: 978-1-78398-024-6   Paperback: 396 pages

89 hands-on recipes to help you complete real-world data science projects in R and Python

1. Learn about the data science pipeline and use it to acquire, clean, analyze, and visualize data.

2. Understand critical concepts in data science in the context of multiple projects.

3. Expand your numerical programming skills through step-by-step code examples and learn more about the robust features of R and Python.

Please check **www.PacktPub.com** for information on our titles