CHAPTER 3

DATA PREPARATION

❖ OBJECTIVES

On completion of this Chapter you will be able to:

- know about the default datasets available in R
- know how to import and export CSV files in R
- know how to import unstructured data files into R
- know how to import XML and HTML files into R
- know how to import JASON and YAML files into R
- know how to import and export excel files in R
- know how to import SAS, SPSS and MATLAB files into R
- know how to import web data files into R
- understand the concept of accessing various databases from R
- manipulate string data
- manipulate data frames
- understand how to melt and cast data in data frames
- understand how the grouping functions are applied on the data in R

3.1. Datasets

R has many datasets built in. R can read data from variety of other data sources and in variety of formats. One of the packages in R is *datasets* which is filled with example datasets. Many other packages also contain datasets. We can see all the datasets available in the loaded packages using the *data()* function.

To access a particular dataset use the *data()* function with its argument as the dataset name enclosed within double quotes and the second optional argument being the package name in which the dataset is present (This second argument is required only if the particular package is not loaded). The invoked dataset can be listed just like a data frame using the *head()* function.

```
> data("kidney", package = "survival")
> head(kidney)
```

	id	time	status	age	sex	disease	frail
1	1	8	1	28	1	Other	2.3
2	1	16	1	28	1	Other	2.3
3	2	23	1	48	2	GN	1.9

....

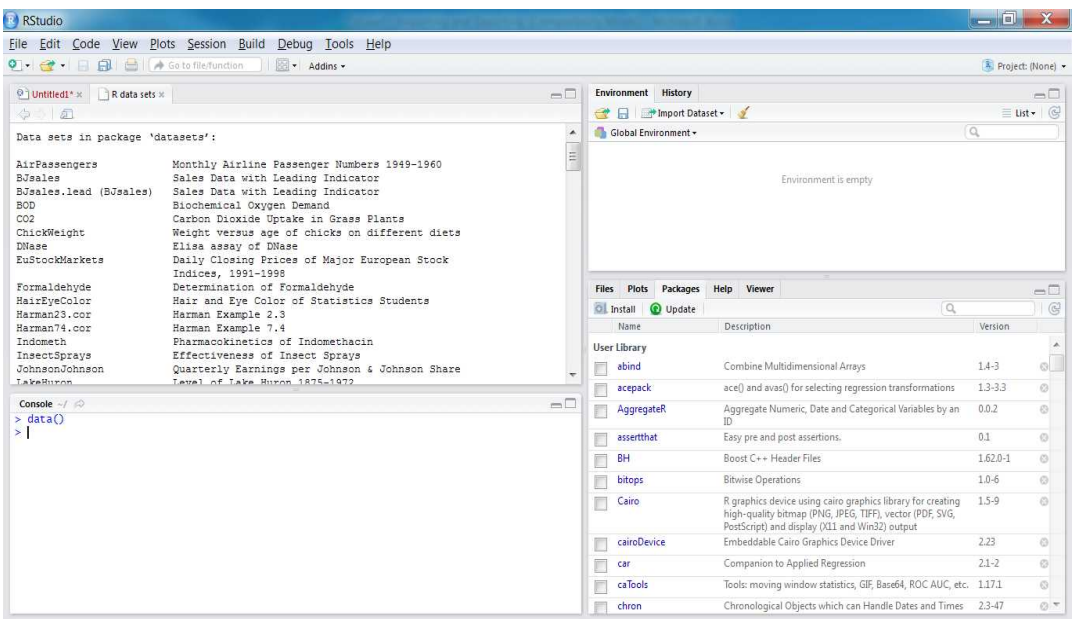

Figure 3.1 R-Studio Showing the List of Datasets

3.2. Importing and Exporting Files

3.2.1. Text and CSV Files

Text documents have several formats. Common format are CSV (Comma Separated Values), XML (Extended Markup Language), JSON (JavaScript Object Notation) and YAML. An example of an unstructured text data is a book.

Comma Separated Values (CSV) Files is a spreadsheet like data stored with comma delimited values. The *read.table()* function reads these files and stores the result in a data frame. If the data has header, it is required to pass the argument *header = TRUE* to the *read.table()* function. The argument *fill = TRUE* makes the *read.table()* function substitute *NA* values for the missing fields. The *system.file()* function is used to locate files that are inside a package. In the below example "*extdata*" is the folder name and the package name is "*learning*" and the file name is "*RedDeerEndocranialVolume,dlm*" The *str()* function takes the data frame name as the argument and lists the structure of the dataset stored in the data frame.

```
> install.packages("learningr")
> library(learningr)
> deer_file <- system.file("extdata","RedDeerEndocranialVolume.dlm",
                                                    package = "learningr")
> deer_data <- read.table(deer_file, header=TRUE, fill=TRUE)
> str(deer_data)
'data.frame':        33 obs. of  8 variables:
$ SkullID    : Factor w/ 33 levels "A4","B11","B12",..: 14 2 17 16 15 13 10 11
                                                                    19 3 ...
 $ VolCT      : int  389 389 352 388 375 325 346 302 379 410 ...
 $ VolBead    : int  375 370 345 370 355 320 335 295 360 400 ...
 $ VolLWH     : int  1484 1722 1495 1683 1458 1363 1250 1011 1621 1740 ...
 $ VolFinarelli: int  337 377 328 377 328 291 289 250 347 387 ...
 $ VolCT2     : int  NA NA NA NA NA NA 346 303 375 413 ...
```

```
$ VolBead2    : int  NA NA NA NA NA NA 330 295 365 395 ...
$ VolLWH2     : int  NA NA NA NA NA NA 1264 1009 1647 1728 ...
```

The column names and row names are listed by default and if the row names are not given in the dataset, the rows are simply numbered 1, 2, 3 and so on. The arguments specify how the file will be read. The argument *sep* determines the character to use as separator between fields. The *nrow* argument specifies the lines of data to read. The argument *skip* specifies the number of lines to skip at the start of the file. For the functions *read.table()* and *read.csv()* the default separator is set to comma and they assume the data has header row. The function *read.csv2()* uses the semicolon as the separator and comma instead of decimals. The *read.delim()* function imports the tab-delimited files with full stops for decimal places. The *read.delim2()* function imports the tab-delimited files with commas for decimal places.

```
> read.csv(deer_file, header=FALSE, skip = 3, nrow = 2)
                                V1
1 DIC90   352      345       1495      328
2 DIC83   388      370       1683      377
> head(deer_data)
   SkullID VolCT VolBead VolLWH VolFinarelli VolCT2 VolBead2 VolLWH2
1   DIC44   389    375   1484        337     NA      NA      NA
2    B11    389    370   1722        377     NA      NA      NA
3   DIC90   352    345   1495        328     NA      NA      NA
....
```

The *colbycol* and *sqldf* packages contain functions that allow to read part of the CSV file into R. These are useful when we don't need all the columns or all the rows. For low-level control we can use the *scan()* function to import CSV file. For data exported from other languages we may need to pass the *na.strings* argument to the *read.table()* function to replace the missing values. If the data is exported from SQL, we use *na.strings* = "*NULL*" and if the data is exported from SAS or Stata, we use *na.strings* = ".". If the data is exported from Excel we use the *na.strings* = *c*("","#N/A", "*#DIV/0!*", "*#NUM!*").

Writing data from R into a file is easier than reading files into R. For this we use the functions *write.table()* and *write.csv()*. These functions take a data frame and a file path as arguments. They also have arguments to specify if we need not include row names in the output file or to specify the character encoding of the output file.

```
> write.csv(deer_data,"F:/deer.csv", row.names = FALSE, fileEncoding = "utf8")
```

3.2.2. Unstructured Files

If the file structure is week, it is easier to read the file as lines of text using the function *readLines()* and then parse the contents. The *readLines()* function accepts a path to the file as the argument. Similarly, the *writeLines()* function takes a text line or a character vector and the file name as argument and writes the text to the file.

```
> tempest <- readLines("F:/Tempest.txt")
> tempest
 [1] "The writing of Prefaces to Plays was probably invented by some very"
 [2] "ambitious Poet, who never thought he had done enough: Perhaps by
ome"
[3] "Ape of the French Eloquence, which uses to make a business of a Letter
of"
....
> writeLines("This book is about a story by Shakespeare", "F:/story.csv")
```

3.2.3. XML and HTML Files

XML files are used for storing nested data. Few of them are RSS (Really Simple Syndication) feeds, SOAP (Simple Object Access Protocols) and XHTML Web Pages. To read the XML files, the *XML* package has to be installed. When an XML file is imported, the result can be stored using the internal nodes or the R nodes. If the result is stored using internal nodes, it allows to query the node tree using the *XPath* language (used for interrogating XML documents). The XML file can be imported using the function *xmlParse()* function. This function can take the argument *useInternalNodes = FALSE* to use R-level nodes instead of the internal nodes while importing the XML files. But, this is set by default by the *xml TreeParse()* function.

```
> install.packages("XML")
> library(XML)

> xml_file <- system.file("extdata", "options.xml", package = "learningr")
> r_options <- xmlParse(xml_file)

> xmlParse(xml_file, useInternalNodes = FALSE)
> xmlTreeParse(xml_file)
```

The functions for importing HTML pages are *htmlParse()* and *htmlTreeParse()* and they behave same as the *xmlParse()* and *xmlTreeParse()* functions.

3.2.4. JASON and YAML Files

The two packages dealing with JSON data are *RJSONIO* and *rjson* and the best of these is the *RJSONIO*. The function used to import the JSON file is *fromJSON()* and the function used to export the JSON file is *toJSON()*. The *yaml* package has two functions for importing YAML data and they are *yaml.load()* and *yaml.load_file()*. The function *as.yaml()* performs the task of converting R objects to YAML strings.

Many softwares store their data in binary formats which are smaller in size than the text files. They hence provide performance gains at the expense of human readability.

3.2.5. Excel Files

Excel is the world's most powerful data analysis tool and its document formats are XLX and XLSX. Spreadsheets can be imported with the functions *read.xlsx()* and *read.xlsx2()*. The *colClasses* argument determines what class each column should have in the resulting data frame and this argument is optional in the above functions. To write to an excel file from R we use the function *write.xlsx2()* that takes the data frame and the file name as arguments. There is another package *xlsReadWrite* that does the same function of the *xlxs* package but this one works only in 32-bit R installations and only on windows.

```
> install.packages("xlsx")
> library(xlsx)
> logfile <- read.xlsx2("F:/Log2015.xls", sheetIndex = 1, startRow = 2, endrow = 72,
        colIndex = 1:5, colClasses = c("character", "numeric", "character",
                                       "character", "integer"))
```

3.2.6. SAS, SPSS and MATLAB Files

The files from a statistical package are imported using the foreign package. The *read.ssd()* function is used to read SAS datasets and the *read.dta()* function is used to read *Stata DTA* files. The *read.spss()* function is used to import the *SPSS data files*. Similarly, these files can be written with the *write.foreign()* function. The *MATLAB binary data files* can be read and written using the *readMat()* and *writeMat()* functions in the *R.matlab* package. The files in picture formats can be read via the *jpeg*, *png*, *tiff*, *rtiff* and *readbitmap* packages.

3.2.7. Web Data

R has ways to import data from web sources using Application Programming Interface (API). For example the World Bank makes its data available using the WDI package and the Polish government data can be accessed using the SmarterPoland package. The twitter package provides access to Twitter's users and their tweet.

The *read.table()* function can accept URL rather than a local file. Accessing a large file from internet can be slow and if the file is required frequently, it is better to download the file using the *download.file()* function and create a local copy and then import that.

```
> cancer_url <- "http://repository.seasr.org/Datasets/UCI/csv/breast-cancer.csv"
> cancer_data <- read.csv(cancer_url)
> str(cancer_data)
'data.frame': 287 obs. of  10 variables:
 $ age       : Factor w/ 7 levels "20-29","30-39",..: 7 3 4 4 3 3 4 4 3 3 ...
 $ menopause : Factor w/ 4 levels "ge40","lt40",..: 4 3 1 1 3 3 3 1 3 3 ...
```

```
$ tumor.size : Factor w/ 12 levels "0-4","10-14",..: 12 3 3 7 7 6 5 8 2 1 ...
$ inv.nodes  : Factor w/ 8 levels "0-2","12-14",..: 8 1 1 1 1 5 5 1 1 1 ...
$ node.caps  : Factor w/ 4 levels "","no","String",..: 3 4 2 2 4 4 2 2 2 2 ...
$ deg.malig  : Factor w/ 4 levels "1","2","3","String": 4 3 1 2 3 2 2 3 2 2 ...
$ breast     : Factor w/ 3 levels "left","right",..: 3 2 2 1 2 1 2 1 1 2 ...
$ breast.quad: Factor w/ 7 levels "","central","left_low",..: 7 4 2 3 3 6 4 4 4 5
$ irradiat   : Factor w/ 3 levels "no","String",..: 2 1 1 1 3 1 3 1 1 1 ...
$ Class      : Factor w/ 3 levels "no-recurrence-events",..: 3 2 1 2 1 2 1 1 1 1

> local_copy <- "cancer.csv"
> download.file(cancer_url, local_copy)
    trying URL 'http://repository.seasr.org/Datasets/UCI/csv/breast-cancer.csv'
    Content type 'application/octet-stream' length 18804 bytes (18 KB)
    downloaded 18 KB

> cancer_data <- read.csv(local_copy)
```

3.3. Accessing Databases

R can connect to all database management systems (DBMS) like *SQLite*, *MySQL*, *MariaDB*, *PostgreSQL* and *Oracle* using the *DBI* package. We need to install and load the *DBI* package and the backend package *RSQLite*. Define a database driver of type *SQLite* using the function *dbDriver()* and setup a connection to the database using the function *dbConnect()*. To retrieve data from the databases you write a query as a string containing SQL commands and send it to the database with the function *dbGetQuery()*.

```
> install.packages("DBI")
> install.packages("RSQLite")
> library(DBI)
> library(RSQLite)
> driver <- dbDriver("SQLite")
```

```
> db_file <- system.file("extdata", "crabtag.sqlite", package = "learningr")
> conn <- dbConnect(driver, db_file)
> query <- "SELECT * FROM IdBlock"
> id_block <- dbGetQuery(conn, query)
> id_block
  Tag ID Firmware Version No Firmware Build Level
1      A03401               2              70
```

Alternatively, the function *dbReadTable()* reads a table from the connected database and the function *dbListTables()* can list all the tables in the database.

```
> dbReadTable(conn, "idblock")
  Tag.ID Firmware.Version.No Firmware.Build.Level
1      A03401               2              70
> dbListTables(conn)
 [1] "Daylog"            "DeploymentNotebook" "IdBlock"
 [4] "LifetimeNotebook"  "TagNotebook"
```

The function *dbDisconnect()* is used for disconnecting and unloading the driver and the function *dbUnloadDriver()* is used to unload the defined database driver.

```
> dbDisconnect(conn)
> dbUnloadDriver(driver)
```

For MySQL database we need to load the *RMySQL* package and set the driver type to be "*MySQL*". The PostgreSQL, Oracle and JDBC databases need the *PostgreSQL*, *ROracle* and *RJDBC* packages respectively. To connect to an SQL Server or Access databases, the *RODBC* package needs to be loaded. In this package, the function *odbcConnect()* is used to connect to the database and the function *sqlQuery()* is used to run a query and the function *odbcClose()* is used to close and cleanup the database connections. There are not much matured methods to access the NoSQL (Not only SQL) databases (lightweight databases – scalable than traditional SQL relational databases). To access the *MongoDB* database the packages *RMongo* and *rmongodb* are used. The database *Cassandra* can be accessed using the package *RCassandra*.

3.4. Data Cleaning and Transforming

3.4.1. Manipulating Stings

In some datasets or data frames logical values are represented as "Y" and "N" instead of *TRUE* and *FALSE*. In such cases it is possible to replace the string with correct logical value as in the example below.

```
> a <- c(1,2,3)
> b <- c("A", "B", "C")
> d <- c("Y", "N", "Y")
> df1 <- data.frame(a, b, d)
> df1
      a  b  d
1     1  A  Y
2     2  B  N
3     3  C  Y
convt <- function(x)
{
 y <- rep.int(NA, length(x))
 y[x == "Y"] <- TRUE
 y[x == "N"] <- FALSE
y
}
> df1$d <- convt(df1$d)
> df1
      a  b  d
1     1  A  TRUE
2     2  B  FALSE
3     3  C  TRUE
```

The functions *grep()* and *grepl()* are used to find a pattern in a given text and the functions *sub()* and *gsub()* are used to replace a pattern with another in a given text. The above four functions belong to the *base* package, but the package *stringr* consists of many such string manipulation functions. The function *str_detect()* in the *stringr* package does the same function of detecting the presence of a given pattern in the given text. We can also use the function *fixed()* to mention if the string that we are searching for is a fixed one.

```
> grep("my", "This is my pen")
[1] 1
> grepl("my", "This is my pen")
[1] TRUE
> sub("my", "your","This is my pen")
[1] "This is your pen"
> gsub("my", "your","This is my pen")
[1] "This is your pen"
> str_detect("This is my pen", "my")
[1] TRUE
> str_detect("This is my pen", fixed("my"))
[1] TRUE
```

In the function *str_detect()*, it is possible to specify the search pattern with a pipe symbol "|" to denote, that we need to find either of the two patterns specified. That is we may be looking for the presence of "," or "*and*" in the given text as shown below.

```
> str_detect("I like mangoes, oranges and pineapples", ",|and")
[1] TRUE
```

The function *str_split()* is used to split a given text based on the pattern specified as below. This function returns a vector. But the function *str_split_fixed()* can be used to split the given text into fixed number of strings based on the specified patterns. This function returns a matrix.

```
> str_split("I like mangoes, oranges and pineapples", ",|and")
[[1]]
[1] "I like mangoes" " oranges "      " pineapples"
> str_split_fixed("I like mangoes, oranges and pineapples", ",|and", n = 3)
     [,1]                  [,2]              [,3]
[1,] "I like mangoes" " oranges " " pineapples"
```

The function *str_count()* can be used to count the number of occurrence of a given pattern in the given text.

```
> str_count("I like mangoes, oranges and pineapples", "a|o")
[1] 6
> str_count("I like mangoes, oranges and pineapples", "s")
[1] 3
```

The function *str_replace()* can be used to replace the specified pattern with another pattern in the given text. This function will only replace the first occurrence of the pattern. Hence, to replace all the occurrences of the pattern we use the function *str_replace_all()*. In these functions, to denote multiple patterns to be replaced, they can be placed within square brackets. This means it should replace all that matches these characters specified within the square brackets.

```
> str_replace("I like mangoes, oranges and pineapples", "s", "sss")
[1] "I like mangoesss, oranges and pineapples"
> str_replace_all("I like mangoes, oranges and pineapples", "s", "sss")
[1] "I like mangoesss, orangesss and pineapplesss"
> str_replace_all("I like mangoes, oranges and pineapples", "[ao]", "-")
[1] "I like m-ng-es, -r-nges -nd pine-pples"
```

In the example below, the various ways of storing the gender values are transformed into one way, ignoring the case differences. This is done using the *str_replace()* function and the *fixed()* functions that ignores the case.

```
> gender <- c("MALE", "Male", "male", "FEMALE", "Female", "female")
> clean_gender <- str_replace(gender, fixed("male", ignore_case = TRUE), "Male")
> clean_gender <- str_replace(clean_gender, fixed("female", ignore_case = TRUE),
                                                                        male")
> clean_gender
[1] "Male"   "Male"   "Male"   "Female" "Female" "Female"
```

3.4.2. Manipulating Data Frames

To add a column to a data frame, we can use the below command to achieve this.

```
> name <- c("Jhon", "Peter", "Mark")
> start_date <- c("1980-10-10", "1999-12-12", "1990-04-05")
> end_date <- c("1989-03-08", "2004-09-20", "2000-09-25")
> service <- data.frame(name, start_date, end_date)
> service
     name       start_date    end_date
1    Jhon       1980-10-10    1989-03-08
2    Peter      1999-12-12    2004-09-20
3    Mark       1990-04-05    2000-09-25
> service$period <- as.Date(service$end_date) - as.Date(service$start_date)
> service
     name       start_date    end_date      period
1    Jhon       1980-10-10    1989-03-08    3071 days
2    Peter      1999-12-12    2004-09-20    1744 days
3    Mark       1990-04-05    2000-09-25    3826 days
```

The same can be achieved using the function *with()* as below.

```
> service$period <- with(service, as.Date(end_date) - as.Date(start_date))
> service
```

	name	start_date	end_date	period
1	Jhon	1980-10-10	1989-03-08	3071 days
2	Peter	1999-12-12	2004-09-20	1744 days
3	Mark	1990-04-05	2000-09-25	3826 days

Another way of doing the same is using the function *within()*. But, the difference lies when there are multiple columns to be added to a data frame, we can easily do the same using the *within()* function in a single command and this is not possible using the *with()* function.

```
> service <- within(service,
  {
    period <- as.Date(end_date) - as.Date(start_date)
    highperiod <- period > 2000
    })
> service
```

	name	start_date	end_date	period	highperiod
1	Jhon	1980-10-10	1989-03-08	3071 days	TRUE
2	Peter	1999-12-12	2004-09-20	1744 days	FALSE
3	Mark	1990-04-05	2000-09-25	3826 days	TRUE

The *mutate()* function in the *plyr* package also does the same function as the function *within()*, but the syntax is slightly different.

```
> library(plyr)
> service <- mutate(service,
  {
    period = as.Date(end_date) - as.Date(start_date)
    highperiod = period > 2000
    })
> service
```

	name	*start_date*	*end_date*	*period*	*highperiod*
1	*Jhon*	*1980-10-10*	*1989-03-08*	*3071 days*	*TRUE*
2	*Peter*	*1999-12-12*	*2004-09-20*	*1744 days*	*FALSE*
3	*Mark*	*1990-04-05*	*2000-09-25*	*3826 days*	*TRUE*

The function *complete.cases()* returns the number of rows in a data frame that is free of missing values. The function *na.omit()* will remove the rows with missing values in a data frame. And the function *na.fail()* throws an error message if the data frame contains any missing values.

```
> crime.data <- read.csv("F:/Crimes.csv")
> nrow(crime.data)
[1] 65535
> complete <- complete.cases(crime.data)
> nrow(crime.data[complete, ])
[1] 63799
> clean.crime.data <- na.omit(crime.data)
> nrow(clean.crime.data)
[1] 63799
```

A data frame can be transformed by choosing few of the columns and ignoring the remaining, but considering all the rows as in the example below.

```
> crime.data <- read.csv("F:/Crimes.csv")
> colnames(crime.data)
 [1] "CASE."                "DATE..OF.OCCURRENCE"  "BLOCK"
 [4] "IUCR"                 "PRIMARY.DESCRIPTION"
"SECONDARY.DESCRIPTION"
 [7] "LOCATION.DESCRIPTION" "ARREST"               "DOMESTIC"
[10] "BEAT"                 "WARD"                 "FBI.CD"
[13] "X.COORDINATE"         "Y.COORDINATE"         "LATITUDE"
[16] "LONGITUDE"            "LOCATION"
```

```
> crime.data1 <- crime.data[, 1:6]
> colnames(crime.data1)
[1] "CASE."                  "DATE..OF.OCCURRENCE"   "BLOCK"
[4] "IUCR"                   "PRIMARY.DESCRIPTION"
"SECONDARY.DESCRIPTION"
```

Alternatively, the data frame can be transformed by selecting only the required rows and retaining all columns of a data frame as in the example below.

```
> nrow(crime.data)
[1] 65535
> crime.data2 <- crime.data[1:10,]
> nrow(crime.data2)
[1] 10
```

The function *sort()* sorts the given vector of numbers or strings. It generally sorts from smallest to largest, but this can be altered using the argument *decreasing = TRUE*.

```
> x <- c(5, 10, 3, 15, 6, 8)
> sort(x)
[1]  3  5  6  8 10 15
> sort(x, decreasing = TRUE)
[1] 15 10  8  6  5  3
> y <- c("X", "AB", "Deer", "For", "Moon")
> sort(y)
[1] "AB"   "Deer" "For"  "Moon" "X"
> sort(y, decreasing = TRUE)
[1] "X"    "Moon" "For"  "Deer" "AB"
```

The function *order()* is the inverse of the *sort()* function. It returns the index of the vector elements in the order as below. But, *x[order(x)]* is same as *sort(x)*. This can be seen by the use of the *identical()* function.

```
> order(x)
[1] 3 1 5 6 2 4
> x[order(x)]
[1]  3  5  6  8 10 15
> identical(sort(x), x[order(x)])
[1] TRUE
```

The *order()* function is more useful than the *sort()* function as it can be used to manipulate the data frames easily.

```
> name <- c("Jhon", "Peter", "Mark")
> start_date <- c("1980-10-10", "1999-12-12", "1990-04-05")
> end_date <- c("1989-03-08", "2004-09-20", "2000-09-25")
> service <- data.frame(name, start_date, end_date)
> service
     name       start_date   end_date
1    Jhon       1980-10-10   1989-03-08
2    Peter      1999-12-12   2004-09-20
3    Mark       1990-04-05   2000-09-25
> startdt <- order(service$start_date)
> service.ordered <- service[startdt, ]
> service.ordered
     name       start_date   end_date
1    Jhon       1980-10-10   1989-03-08
3    Mark       1990-04-05   2000-09-25
2    Peter      1999-12-12   2004-09-20
```

The *arrange()* function of the *plyr* package does the same function as above.

```
> library(plyr)
> arrange(service, start_date)
```

	name	start_date	end_date
1	Jhon	1980-10-10	1989-03-08
2	Mark	1990-04-05	2000-09-25
3	Peter	1999-12-12	2004-09-20

The *rank()* function lists the rank of the elements in a vector or a data frame. By specifying the argument *ties.method = "first"*, a rank need not be shared among more than one element with the same value.

```
> x <- c(9, 5, 4, 6, 4, 5)
> rank(x)
[1] 6.0 3.5 1.5 5.0 1.5 3.5
> rank(x, ties.method = "first")
[1] 6 3 1 5 2 4
```

The SQL statements can be executed from R and the results can be obtained as in any other database. The package *sqldf* needs to be installed to manipulate the data frames or datasets using SQL.

```
> install.packages("sqldf")
> library(sqldf)
> query <- "SELECT * FROM iris WHERE Species = 'setosa'"
> sqldf(query)
```

3.4.3. Data Reshaping

Data Reshaping in R is about changing the way data is organized into rows and columns. Most of the time data processing in R is done by taking the input data as a data frame. It is easy to extract data from the rows and columns of a data frame. But there are situations when we need the data frame in a different format than what we received. R has few functions to split, merge and change the columns to rows and vice-versa in a data frame.

The *cbind()* function can be used to join multiple vectors to create a data frame. We can also merge two data frames using the *rbind()* function.

```
> city <- c("Tampa","Seattle","Hartford","Denver")
> state <- c("FL","WA","CT","CO")
> zipcode <- c(33602,98104,06161,80294)
> addresses <- cbind(city,state,zipcode)
> addresses
     city       state      zipcode
[1,] "Tampa"    "FL"       "33602"
[2,] "Seattle"  "WA"       "98104"
[3,] "Hartford" "CT"       "6161"
[4,] "Denver"   "CO"       "80294"
> new.address <- data.frame(
+   city = c("Lowry","Charlotte"),
+   state = c("CO","FL"),
+   zipcode = c("80230","33949"),
+   stringsAsFactors = FALSE
+ )
> print(new.address)
     city       state      zipcode
1    Lowry      CO         80230
2    Charlotte  FL         33949
> all.addresses <- rbind(addresses,new.address)
> all.addresses
     city       state      zipcode
1    Tampa      FL         33602
2    Seattle    WA         98104
3    Hartford   CT         6161
4    Denver     CO         80294
```

5	*Lowry*	*CO*	*80230*
6	*Charlotte*	*FL*	*33949*

The *merge()* function can be used to merge two data frames. The merging requires the data frames to have same column names on which the merging is done. In the example below, we consider the data sets about *Diabetes in Pima Indian Women* available in the library named "*MASS*". The two datasets are merged based on the values of *blood pressure* ("*bp*") and *body mass index* ("*bmi*"). On choosing these two columns for merging, the records where values of these two variables match in both data sets are combined together to form a single data frame.

```
> library(MASS)
> head(Pima.te)
```

	npreg	*glu*	*bp*	*skin*	*bmi*	*ped*	*age*	*type*
1	*6*	*148*	*72*	*35*	*33.6*	*0.627*	*50*	*Yes*
2	*1*	*85*	*66*	*29*	*26.6*	*0.351*	*31*	*No*
3	*1*	*89*	*66*	*23*	*28.1*	*0.167*	*21*	*No*
...								

```
> head(Pima.tr)
```

	npreg	*glu*	*bp*	*skin*	*bmi*	*ped*	*age*	*type*
1	*5*	*86*	*68*	*28*	*30.2*	*0.364*	*24*	*No*
2	*7*	*195*	*70*	*33*	*25.1*	*0.163*	*55*	*Yes*
3	*5*	*77*	*82*	*41*	*35.8*	*0.156*	*35*	*No*
...								

```
> nrow(Pima.te)
[1] 332
> nrow(Pima.tr)
[1] 200
> merged.Pima <- merge(x = Pima.te, y = Pima.tr,
+                      by.x = c("bp", "bmi"),
+                      by.y = c("bp", "bmi")
```

```
+ )
> head(merged.Pima)
  bp  bmi npreg.x glu.x skin.x ped.x age.x type.x npreg.y glu.y skin.y
1 60 33.8      1   117     23 0.466    27     No       2   125     20
2 64 29.7      2    75     24 0.370    33     No       2   100     23
3 64 31.2      5   189     33 0.583    29    Yes       3   158     13
...
  ped.y age.y type.y
1 0.088    31     No
2 0.368    21     No
3 0.295    24     No
...
> nrow(merged.Pima)
[1] 17
```

One of the most interesting aspects of R programming is about changing the shape of the data in multiple steps to get a desired shape. The functions used to do this are called *melt()* and *cast()*. We consider the dataset called *ships* present in the library called "*MASS*".

```
> library(MASS)
> head(ships)
  type year period service incidents
1    A   60     60     127         0
2    A   60     75      63         0
3    A   65     60    1095         3
...
```

Now we melt the data using the *melt()* function in the package *reshape2* to organize it, converting all columns other than type and year into multiple rows.

```
> library(reshape2)
> molten.ships <- melt(ships, id = c("type","year"))
> head(molten.ships)
```

	type	year	variable	value
1	A	60	period	60
2	A	60	period	75
3	A	65	period	60
4	A	65	period	75
5	A	70	period	60
6	A	70	period	75

```
> nrow(molten.ships)
[1] 120
> nrow(ships)
[1] 40
```

We can cast the molten data into a new form where the aggregate of each type of ship for each year is created. It is done using the *cast()* function.

```
> recasted.ship <- cast(molten.ships, type+year~variable,sum)
> head(recasted.ship)
```

	type	year	period	service	incidents
1	A	60	135	190	0
2	A	65	135	2190	7
3	A	70	135	4865	24
4	A	75	135	2244	11
5	B	60	135	62058	68
6	B	65	135	48979	111

3.4.4. Grouping Functions

R has many apply functions such as *apply()*, *lapply()*, *sapply()*, *vapply()*, *mapply()*, *rapply()*, *tapply()*, *aggregate()* and *by()*. Function *lapply()* is a list apply which acts on a list or vector and returns a list. Function *sapply()* is a simple *lapply()* function defaults to returning a vector or matrix when possible. Function *vapply()* is a verified *apply()* function that allows the return object type to be pre-specified. Function *rapply()* is a recursive apply for nested lists, i.e. lists within lists. Function *tapply()* is a tagged apply where the tags identify the subsets. Function *apply()* is generic, applies a function to a matrix's rows or columns or, more generally, to dimensions of an array.

If we want to apply a function to the rows or columns of a matrix or array, we use the *apply()* function as below.

```
> M <- matrix(seq(1,16), 4, 4)
> M
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

> apply(M, 1, min)
[1] 1 2 3 4

> apply(M, 2, max)
[1]  4  8 12 16
> M <- array( seq(32), dim = c(4,4,2))
> M
, , 1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
, , 2
     [,1] [,2] [,3] [,4]
[1,]   17   21   25   29
[2,]   18   22   26   30
[3,]   19   23   27   31
[4,]   20   24   28   32

> apply(M, 1, sum)
[1] 120 128 136 144

> apply(M, c(1,2), sum)
     [,1] [,2] [,3] [,4]
[1,]   18   26   34   42
[2,]   20   28   36   44
[3,]   22   30   38   46
[4,]   24   32   40   48
```

If we want to apply a function to each element of a list in turn and get a list back, we use the *lapply()* function as below.

```
> x <- list(a = 1, b = 1:3, c = 10:100)
> x
$a
[1] 1
$b
[1] 1 2 3
```

```
$c
 [1]  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26
[18]  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43
[35]  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
[52]  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77
[69]  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94
[86]  95  96  97  98  99 100
```

```
> lapply(x, FUN = length)
$a
[1] 1
$b
[1] 3
$c
[1] 91
```

```
> lapply(x, FUN = sum)
$a
[1] 1
$b
[1] 6
$c
[1] 5005
```

We use the function *sapply()*, if we want to apply a function to each element of a list in turn, and we want a vector back.

```
> x <- list(a = 1, b = 1:3, c = 10:100)
> x
$a
[1] 1
```

```
$b
[1] 1 2 3
$c
 [1]  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26
[18]  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43
[35]  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
[52]  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77
[69]  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94
[86]  95  96  97  98  99 100
```

```
> sapply(x, FUN = length)
 a  b  c
 1  3 91
> sapply(x, FUN = sum)
   a   b    c
   1   6 5005
```

When we want to use the function *sapply()*, but need to squeeze some more speed out of the code, we use the function *vapply()* as below. For the function *vapply()*, we give R the information on what the function will return, which can save some time coercing returned values to fit in a single atomic vector. In the example below, we tell R that everything returned by *length()* should be an integer of length 1.

```
> x <- list(a = 1, b = 1:3, c = 10:100)
> vapply(x, FUN = length, FUN.VALUE = 0L)
 a  b  c
 1  3 91
```

For when we have several data structures (e.g. vectors, lists) and we want to apply a function to the 1st elements of each, and then the 2nd elements of each, etc., coercing the result to a vector/array we use the function *vapply()* as below.

```
> mapply(sum, 1:5, 1:5, 1:5)
[1]  3  6  9 12 15
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

When we want to apply a function to each element of a nested list structure, recursively, we use the function *rapply()* as below. The function *rapply()* can be best illustrated with a user defined function to be applied.

```
> myFun <- function(x) {
+    if (is.character(x)) {
+       return(paste(x,"!",sep=""))
+    }
+    else{
+       return(x + 1)
+    }
+ }

> l <- list(a = list(a1 = "Boo", b1 = 2, c1 = "Eeek"),
+          b = 3, c = "Yikes",
+          d = list(a2 = 1, b2 = list(a3 = "Hey", b3 = 5)))

> rapply(l, myFun)
```

```
  a.a1     a.b1     a.c1     b      c          d.a2   d.b2.a3      d.b2.b3
"Boo!"    "3"    "Eeek!"   "4"   "Yikes!"    "2"    "Hey!"       "6"
> rapply(l, myFun, how = "replace")
$a
$a$a1
[1] "Boo!"
$a$b1
[1] 3
$a$c1
[1] "Eeek!"
$b
[1] 4
$c
[1] "Yikes!"
$d
$d$a2
[1] 2
$d$b2
$d$b2$a3
[1] "Hey!"
$d$b2$b3
[1] 6
```

When we want to apply a function to subsets of a vector and the subsets are defined by some other vector, usually a factor, we use the function *tapply()* as below.

```
> x <- 1:20
> x
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
> y <- factor(rep(letters[1:5], each = 4))
> y
[1] a a a a b b b b c c c c d d d d e e e e
Levels: a b c d e

> tapply(x, y, sum)
 a  b  c  d  e
10 26 42 58 74
```

The *by()* function, can be thought of, as a "wrapper" for the function *tapply()*. When we want to compute a task that *tapply()* can't handle, the *by()* function arises.

```
> cta <- tapply(iris$Sepal.Width , iris$Species , summary )
> cba <- by(iris$Sepal.Width , iris$Species , summary )
> cta
$setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.300   3.200   3.400   3.428   3.675   4.400
$versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.525   2.800   2.770   3.000   3.400
$virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.200   2.800   3.000   2.974   3.175   3.800
> cba
iris$Species: setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.300   3.200   3.400   3.428   3.675   4.400
   ------------------------------------------------------
```

```
iris$Species: versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.000   2.525   2.800   2.770   3.000   3.400
   ------------------------------------------------------
iris$Species: virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.200   2.800   3.000   2.974   3.175   3.800
```

If we print these two objects, *cta* and *cba*, we have the same results. The only differences are in how they are shown with the different class attributes. The power of the function *by()* arises when we can't use the function *tapply()* as in the following code.

```
> tapply(iris, iris$Species, summary )
Error in tapply(iris, iris$Species, summary) :
  arguments must have same length
```

R says that arguments must have the same lengths, say "we want to calculate the summary of all variable in iris along the factor Species": but R just can't do that because it does not know how to handle. The *by()* function lets the *summary()* function work even if the length of the first argument are different.

```
> bywork <- by(iris, iris$Species, summary )
> bywork
iris$Species: setosa
    Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
   Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100
   1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200
   Median :5.000   Median :3.400   Median :1.500   Median :0.200
   Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
   3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
   Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
```

```
    Species
    setosa    :50
    versicolor: 0
    virginica : 0
    ------------------------------------------------------
iris$Species: versicolor
  Sepal.Length    Sepal.Width     Petal.Length   Petal.Width
  Min.   :4.900   Min.   :2.000   Min.   :3.00   Min.   :1.000
  1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200
  Median :5.900   Median :2.800   Median :4.35   Median :1.300
  Mean   :5.936   Mean   :2.770   Mean   :4.26   Mean   :1.326
  3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
  Max.   :7.000   Max.   :3.400   Max.   :5.10   Max.   :1.800
    Species
    setosa    : 0
    versicolor:50
    virginica : 0
    ------------------------------------------------------
iris$Species: virginica
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
  Min.   :4.900   Min.   :2.200   Min.   :4.500   Min.   :1.400
  1st Qu.:6.225   1st Qu.:2.800   1st Qu.:5.100   1st Qu.:1.800
  Median :6.500   Median :3.000   Median :5.550   Median :2.000
  Mean   :6.588   Mean   :2.974   Mean   :5.552   Mean   :2.026
  3rd Qu.:6.900   3rd Qu.:3.175   3rd Qu.:5.875   3rd Qu.:2.300
  Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
    Species
    setosa    : 0
```

```
versicolor: 0
virginica :50
```

The arguments must have the same lengths. R can't do that because it does not know how to handle it. The *by()* function lets the *summary()* function work even if the length of the first argument is different. The result is an object of class by that along Species computes the summary of each variable.

The *aggregate()* function can be seen as another a different way of using *tapply()* function if we use it in such a way.

```
> att <- tapply(iris$Sepal.Length , iris$Species , mean)
> agt <- aggregate(iris$Sepal.Length , list(iris$Species), mean)
> att
    setosa versicolor  virginica
    5.006      5.936      6.588
> agt
         Group.1        x
1        setosa        5.006
2    versicolor        5.936
3     virginica        6.588
```

The two immediate differences are that the second argument of the *aggregate()* function must be a list while *tapply()* function can (not mandatory) be a list and that the output of the *aggregate()* function is a data frame while the one of *tapply()* function is an array. The power of the *aggregate()* function is that it can handle easily subsets of the data with *subset* argument and that it can handle formula as well. These elements make the *aggregate()* function easier to work with than *tapply()* function in some situations.

```
> ag <- aggregate(len ~ ., data = ToothGrowth, mean)
> ag
```

	supp	*dose*	*len*
1	*OJ*	*0.5*	*13.23*
2	*VC*	*0.5*	*7.98*
3	*OJ*	*1.0*	*22.70*
4	*VC*	*1.0*	*16.77*
5	*OJ*	*2.0*	*26.06*
6	*VC*	*2.0*	*26.14*

❖ HIGHLIGHTS

- One of the packages in R is *datasets* which is filled with example datasets.
- We can see all the datasets available in the loaded packages using the *data()* function.
- The *read.table()* function reads the CSV files and stores the result in a data frame.
- The *system.file()* function is used to locate files that are inside a package.
- Writing data from R into a file is done using the functions *write.table()* and *write.csv()*.
- If the file is unstructured, it is read using the function *readLines()*.
- The *writeLines()* function takes a text line and the file name as argument and writes the text to the file.
- The XML file can be imported using the function *xmlParse()* function.
- The function used to import the JSON file is *fromJSON()* and the function used to export the JSON file is *toJSON()*.
- Spreadsheets can be imported with the functions *read.xlsx()* and *read.xlsx2()*.
- To write to an excel file from R we use the function *write.xlsx2()*.
- The *read.ssd()* function is used to read SAS datasets.
- The *read.spss()* function is used to import the SPSS data files.
- The *MATLAB* binary data files can be read and written using the *readMat()* and *writeMat()* functions in the *R.matlab* package.

- R can connect to all DBMS like SQLite, MySQL, MariaDB, PostgreSQL and Oracle using the DBI package.
- The function *dbReadTable()* reads a table from the connected database.
- The functions *grep()* and *grepl()* are used to find a pattern in a given text.
- The functions *sub()* and *gsub()* are used to replace a pattern with another in a given text.
- The function *str_split()* is used to split a given text based on the pattern specified.
- The function *str_count()* can be used to count the number of occurrence of a given pattern in the given text.
- The function *str_replace()* can be used to replace the specified pattern with another pattern in the given text.
- The function *na.omit()* will remove the rows with missing values in a data frame.
- The function *sort()* sorts the given vector of numbers or strings.
- The function *order()* is the inverse of the *sort()* function.
- The *rank()* function lists the rank of the elements in a vector or a data frame.
- The package *sqldf* needs to be installed to manipulate the data frames or datasets using SQL.
- Changing the shape of a data frame is done using the functions *melt()* and *cast()*.
- R has many grouping functions such as *apply()*, *lapply()*, *sapply()*, *vapply()*, *mapply()*, *rapply()*, *tapply()*, *aggregate()* and *by()*.