Beginning\_R\_Joshua\_C02

**Chapter 2**

**Dealing with Dates, Strings, and**

**Data Frames**

The world of data and data analytics is changing rapidly. Data analysts are facing major issues related to

the use of larger datasets, including cloud computing and the creation of so-called data lakes, which are

enterprise-wide data management platforms consisting of vast amounts of data in their original format

stored in an single unmanaged and unstructured location available to the entire organization. This flies in

the face of the carefully structured and highly managed data most of us have come to know and love.

Data lakes solve the problem of independently managed information silos (an old problem in

information technology), and the newer problem of dealing with Big Data projects, which typically require

large amounts of highly varied data. If you are particularly interested in using R for cloud computing, I

recommend Ajay Ohri’s book R for Cloud Computing: An Approach for Data Scientists. We will

touch lightly on the issues of dealing with R in the cloud and with big (or at least bigger) data in subsequent

chapters.

You learned about various data types in Chapter 1. To lay the foundation for discussing some ways

of dealing with real-world data effectively, we first discuss working with dates and times and then discuss

working with data frames in more depth. In later chapters, you will learn about data tables, a package that

provides a more efficient way to work with large datasets in R.

2.1 Working with Dates and Times

Dates and times are handled differently by R than other data. Dates are represented as the number of days

since January 1, 1970, with negative numbers representing earlier dates. You can return the current date and

time by using the date() function and the current day by using the Sys.Date() function:

> date ()

[1] "Fri Dec 26 07:00:28 2014 "

> Sys . Date ()

[1] " 2014 -12 -26 "

By adding symbols and using the format() command, you can change how dates are shown.

These symbols are as follows:

• %d The day as a number

• %a Abbreviated week day

• %A Unabbreviated week day

Chapter 2 ■ Dealing with Dates, Strings, and Data Frames

16

• %b Abbreviated month

• %B Unabbreviated month

• %y Two-digit year

• %Y Four-digit year

See the following example run by the author on 1 January 2015. Notice the use of cat() to concatenate

and output the desired objects:

> today <- Sys . Date ()

> cat ( format (today , format = "%A, %B %d, %Y")," Happy New Year !", "\n")

Thursday , January 01, 2015 Happy New Year !

2.2 Working with Strings

You have already seen character data, but let’s spend some time getting familiar with how to manipulate

strings in R. This is a good precursor to our more detailed discussion of text mining later on. We will look

at how to get string data into R, how to manipulate such data, and how to format string data to maximum

advantage. Let’s start with a quote from a famous statistician, R. A. Fisher:

The null hypothesis is never proved or established, but is possibly disproved, in the course

of experimentation. Every experiment may be said to exist only to give the facts a chance of

disproving the null hypothesis.” R. A. Fisher

Although it would be possible to type this quote into R directly using the console or the R Editor, that

would be a bit clumsy and error-prone. Instead, we can save the quote in a plain text file. There are many

good text editors, and I am using Notepad++. Let’s call the file “fishersays.txt” and save it in the current

working directory:

> dir ()

[1] " fishersays . txt " " mouse \_ weights \_ clean . txt"

[3] " mouseSample . csv " " mouseWts . rda "

[5] " zScores . R"

You can read the entire text file into R using either readLines() or scan(). Although scan() is more

flexible, in this case a text file consisting of a single line of text with a “carriage return” at the end is very easy

to read into R using the readLines() function:

> fisherSays <- readLines ("fishersays.txt")

> fisherSays

[1] "The null hypothesis is never proved or established , but is possibly disproved ,

in the course of experimentation . Every experiment may be said to exist only to

give the facts a chance of disproving the null hypothesis . R. A. Fisher "

>

Note that I haven’t had to type the quote at all. I found the quote on a statistics quotes web page, copied

it, saved it into a text file, and then read it into R.

As a statistical aside, Fisher’s formulation did not (ever) require an alternative hypothesis. Fisher was

a staunch advocate of declaring a null hypothesis that stated a certain population state of affairs, and then

determining the probability of obtaining the sample results (what he called facts), assuming that the null

hypothesis was true. Thus, in Fisher’s formulation, the absence of an alternative hypothesis meant that

Type II errors were simply ignored, whereas Type I errors were controlled by establishing a reasonable

significance level for rejecting the null hypothesis. We will have much more to discuss about the current state

and likely future state of null hypothesis significance testing (NHST), but for now, let’s get back to strings.

A regular expression is a specific pattern in a string or a set of strings. R uses three types of such

expressions:

• Regular expressions

• Extended regular expressions

• Perl-like regular expressions

The functions that use regular expressions in R are as follows (see Table 2-1). You can also use the

glob2rx() function to create specific patterns for use in regular expressions. In addition to these functions,

there are many extended regular expressions, too many to list here. We can search for specific characters,

digits, letters, and words. We can also use functions on character strings as we do with numbers, including

counting the number of characters, and indexing them as we do with numbers. We will continue to work

with our quotation, perhaps making Fisher turn over in his grave by our alterations.

Table 2-1. R Functions that use regular expressions

Purpose Function Explanation

Substitution sub() Both sub() and gsub() are used to make substitutions in a string

Extraction grep() Extract some value from a string

Detection grepl() Detect the presence of a pattern

The simplest form of a regular expression are ones that match a single character. Most characters,

including letters and digits, are also regular expressions. These expressions match themselves. R also

includes special reserved characters called metacharacters in the extended regular expressions. These have

a special status, and to use them, you must use a double backslash \\to escape these when you need to use

them as literal characters. The reserved characters are ., \, |, (, ), [, {, $, \*, +, and ?.

Let us pretend that Jerzy Neyman actually made the quotation we attributed to Fisher. This is certainly

not true, because Neyman and Egon Pearson formulated both a null and an alternative hypothesis and

computed two probabilities rather than one, determining which hypothesis had the higher probability of

having generated the sample data. Nonetheless, let’s make the substitution. Before we do, however, look at

how you can count the characters in a string vector. As always, a vector with one element has an index of [1],

but we can count the actual characters using the nchar() function:

> length ( fisherSays )

[1] 1

> nchar ( fisherSays )

[1] 230

sub ("R. A. Fisher", "Jerzy Neyman", fisherSays )

[1] "The null hypothesis is never proved or established, but is possibly disproved, in the

course of experimentation. Every experiment may be said to exist only to give the facts a

chance of disproving the null hypothesis." Jerzy Neyman"

2.3 Working with Data Frames in the Real World

Data frames are the workhorse data structure for statistical analyses. If you have used other statistical

packages, a data frame will remind you of the data view in SPSS or of a spreadsheet. Customarily, we use

columns for variables and rows for units of analysis (people, animals, or objects). Sometimes we need to

change the structure of the data frame to accommodate certain situations, and you will learn how to stack

and unstack data frames as well as how to recode data when you need to.

There are many ways to create data frames, but for now, let’s work through a couple of data frames

built into R. The data frame comes from the 1974 Motor Trend US Magazine, and contains miles per gallon,

number of cylinders, displacement, gross horsepower, rear axle ratio, weight, quarter mile time in seconds,

‘V’ or Straight engine, transmission, number of forward gears, and number of carburetors.

The complete dataset has 32 cars and 10 variables for each car. We will also learn how to find specific

rows of data:

> str(mtcars)

'data.frame': 32 obs. of 11 variables:

$ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...

$ cyl : num 6 6 4 6 8 6 8 4 4 6 ...

$ disp: num 160 160 108 258 360 ...

$ hp : num 110 110 93 110 175 105 245 62 95 123 ...

$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...

$ wt : num 2.62 2.88 2.32 3.21 3.44 ...

$ qsec: num 16.5 17 18.6 19.4 17 ...

$ vs : num 0 0 1 1 0 1 0 1 1 1 ...

$ am : num 1 1 1 0 0 0 0 0 0 0 ...

$ gear: num 4 4 4 3 3 3 3 4 4 4 ...

$ carb: num 4 4 1 1 2 1 4 2 2 4 ...

> summary(mtcars $ mpg)

Min. 1st Qu. Median Mean 3rd Qu. Max.

10.40 15.42 19.20 20.09 22.80 33.90

> summary(mtcars $ wt)

Min. 1st Qu. Median Mean 3rd Qu. Max.

1.513 2.581 3.325 3.217 3.610 5.424

To refer to a given column in a data fame, you can use either indexing or the $ operator with the data

frame name followed by the variable name. Because data frames have both rows and columns, you must

use indexes for both the row and the column. To refer to an entire row or an entire column, you can use

a comma, as you can with a matrix. To illustrate, the rear axle ratio variable is the fifth column in the data

frame. We can refer to this column in two ways. We can use the dataset$variable notation mtcars $ drat,

or we can equivalently use matrix-type indexing, as in [, 5] using the column number. The head() function

returns the first part or parts of a vector, matrix, data frame, or function, and is useful for a quick “sneak

preview”:

> head( mtcars $ drat)

[1] 3.90 3.90 3.85 3.08 3.15 2.76

> head( mtcars [,5] )

[1] 3.90 3.90 3.85 3.08 3.15 2.76

2.3.1 Finding and Subsetting Data

Sometimes, it is helpful to locate in which row a particular set of data may be. We can find the row containing

a particular value very easily using the which() function:

> which ( mtcars $ hp >= 300)

[1] 31

> mtcars [31 ,]

mpg cyl disp hp drat wt qsec vs am gear carb

Maserati Bora 15 8 301 335 3.54 3.57 14.6 0 1 5 8

Suppose the Maserati’s horsepower need to be recoded to NA because it turns out there was an error in

recording the data (note: this occurs on occasion in real world data), just do the following:

mtcars $ hp [ mtcars $ hp >= 300] <- NA

> mtcars [31 ,]

mpg cyl disp hp drat wt qsec vs am gear carb

Maserati Bora 15 8 301 NA 3.54 3.57 14.6 0 1 5 8

With the one observation recoded to missing, a histogram of the horsepower data is shown

(see Figure 2-1):

Figure 2-1. Car horsepower (with Maserati removed) vs frequency

The data frame indexing using square brackets is similar to that of a matrix. As with vectors, we can

use the colon separator to refer to ranges of columns or rows. For example, say that we are interested in

reviewing the car data for vehicles with manual transmission. Here is how to subset the data in R. Attaching

the data frame makes it possible to refer to the variable names directly, and thus makes the subsetting

operation a little easier. As you can see, the resulting new data frame contains only the manual transmission

vehicles:

> attach ( mtcars )

> mpgMan <- subset ( mtcars , am == 1, select = mpg : disp )

> summary ( mpgMan $ mpg)

Min. 1st Qu. Median Mean 3rd Qu. Max.

15.00 21.00 22.80 24.39 30.40 33.90

You can remove a column in a data frame by assigning it the special value NULL. For this illustration, let

us use a small sample of the data. We will remove the displacement variable. First, recall the data frame:

> mpgMan

mpg cyl disp

Mazda RX4 21.0 6 160.0

Mazda RX4 Wag 21.0 6 160.0

Datsun 710 22.8 4 108.0

Fiat 128 32.4 4 78.7

Honda Civic 30.4 4 75.7

Toyota Corolla 33.9 4 71.1

Fiat X1-9 27.3 4 79.0

Porsche 914-2 26.0 4 120.3

Lotus Europa 30.4 4 95.1

Ford Pantera L 15.8 8 351.0

Ferrari Dino 19.7 6 145.0

Maserati Bora 15.0 8 301.0

Volvo 142E 21.4 4 121.0

Now, simply type the following to remove the variable, and note that the disp variable is no longer part

of the data frame. Also, don’t try this at home unless you make a backup copy of your important data first.

> mpgMan $ disp <- NULL

> mpgMan

mpg cyl

Mazda RX4 21.0 6

Mazda RX4 Wag 21.0 6

Datsun 710 22.8 4

Fiat 128 32.4 4

Honda Civic 30.4 4

Toyota Corolla 33.9 4

Fiat X1-9 27.3 4

Porsche 914-2 26.0 4

Lotus Europa 30.4 4

Ford Pantera L 15.8 8

Ferrari Dino 19.7 6

Maserati Bora 15.0 8

Volvo 142E 21.4 4

We can add a new variable to a data frame simply by creating it, or by using the cbind() function.

Here’s a little trick to make up some data quickly. I used the rep() function (for replicate) to generate 15

“observations” of the color of the vehicle. First, I created a character vector with three color names, then

I replicated the vector five times to fabricate my new variable. By defining it as mpgMan$colors, I was able

to create it and add it to the data frame at the same time. Notice I only used the first 13 entries of colors as

mpgMan only has 13 manual vehicles:

colors <- c(" black ", " white ", " gray ")

> colors <- rep (colors, 5)

> mpgMan $ colors <- colors[1:13]

> mpgMan

mpg cyl colors

Mazda RX4 21.0 6 black

Mazda RX4 Wag 21.0 6 white

Datsun 710 22.8 4 gray

Fiat 128 32.4 4 black

Honda Civic 30.4 4 white

Toyota Corolla 33.9 4 gray

Fiat X1-9 27.3 4 black

Porsche 914-2 26.0 4 white

Lotus Europa 30.4 4 gray

Ford Pantera L 15.8 8 black

Ferrari Dino 19.7 6 white

Maserati Bora 15.0 8 gray

Volvo 142E 21.4 4 black

2.4 Manipulating Data Structures

Depending on the required data analysis, we sometimes need to restructure data by changing narrow format

data to wide-format data, and vice versa. Let’s take a look at some ways data can be manipulated in R. Wide

and narrow data are often referred to as unstacked and stacked, respectively. Both can be used to display

tabular data, with wide data presenting each data value for an observation in a separate column. Narrow

data, by contrast, present a single column containing all the values, and another column listing the “context”

of each value. Recall our roster data from Chapter 1.

It is easier to show this than it is to explain it. Examine the following code listing to see how this works.

We will start with a narrow or stacked representation of our data, and then we will unstack the data into the

more familiar wide format:

> roster <- read.csv("roster.csv")

> sportsExample <- c("Jersey", "Class")

> stackedData <- roster [ sportsExample ]

> stackedData

Jersey Class

1 0 freshman

2 1 sophomore

3 3 junior

4 5 sophomore

5 10 freshman

6 12 senior

7 15 freshman

8 20 junior

9 21 senior

10 33 junior

11 35 junior

12 44 junior

13 50 sophomore

> unstack(stackedData)

$freshman

[1] 0 10 15

$junior

[1] 3 20 33 35 44

$senior

[1] 12 21

$sophomore

[1] 1 5 50

2.5 The Hard Work of Working with Larger Datasets

As I have found throughout my career, real-world data present many challenges. Datasets often have missing

values and outliers. Real data distributions are rarely normally distributed. The majority of the time I have

spent with data analysis has been in preparation of the data for subsequent analyses, rather than the analysis

itself. Data cleaning and data munging are rarely included as a subject in statistics classes, and included

datasets are generally either fabricated or scrubbed squeaky clean.

The General Social Survey (GSS) has been administered almost annually since 1972. One commentator

calls the GSS “America’s mood ring.” The data for 2012 contain the responses to a 10-word vocabulary test.

Each correct and incorrect responses are labeled as such, with missing data coded as NA. The GSS data are

available in SPSS and STATA format, but not in R format. I downloaded the data in SPSS format and then use

the R library foreign to read that into R as follows. As you learned earlier, the View function allows you to see

the data in a spreadsheet-like layout (see Figure 2-2):

> library(foreign)

> gss2012 <- read.spss("GSS2012merged\_R5.sav", to.data.frame = TRUE)

> View(gss2012)

Here’s a neat trick: The words are in columns labeled “worda”, “wordb”, . . . , “wordj”. I want to subset

the data, as we discussed earlier, to keep from having to work with the entire set of 1069 variables and 4820

observations. I can use R to make my list of variable names without having to type as much as you might

suspect. Here’s how I used the paste0 function and the built-in letters function to make it easy. There is

an acronym among computer scientists called DRY that was created by Andrew Hunt and David Thomas:

“Don’t repeat yourself.” According to Hunt and Thomas, pragmatic programmers are early adopters, fast

adapters, inquisitive, critical thinkers, realistic, and jacks of all trades:

> myWords <- paste0 ("word", letters [1:10])

> myWords

[1] "worda" "wordb" "wordc" "wordd" "worde" "wordf" "wordg" "wordh" "wordi" "wordj"

> vocabTest <- gss2012 [ myWords ]

> head ( vocabTest )

worda wordb wordc wordd worde wordf wordg wordh wordi wordj

1 CORRECT CORRECT INCORRECT CORRECT CORRECT CORRECT INCORRECT INCORRECT CORRECT CORRECT

2 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>

3 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>

4 CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT INCORRECT

5 CORRECT CORRECT INCORRECT CORRECT CORRECT CORRECT INCORRECT <NA> CORRECT INCORRECT

6 CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT CORRECT <NA> CORRECT INCORRECT

We will also apply the DRY principle to our analysis of our subset data. For each of the words, it would

be interesting to see how many respondents were correct versus incorrect. This is additionally interesting

because we have text rather than numerical data (a frequent enough phenomena in survey data). There are

many ways perhaps to create the proportions we seek, but let us explore one such path. Of note here is that

we definitely recommend using the top left Rscript area of Rstudio to type in these functions, then selecting

that code and hitting <Ctrl> + R to run it all in the console.

First, some exploration of a few new functions. The table() function creates a contingency table with

a count of each combination of factors. Secondly, note the output of myWords[1]. Keeping in mind the

DRY principle, notice the difference between our first use of table versus the second use. It seems a little

changed, no? And yet, if we wanted to get counts for each of our words a through j, the second is much more

powerful if we could simply find a way to increase that counter by 1 each time we ran the code.

> myWords[1]

[1] "worda"

> table(vocabTest[, "worda"], useNA = "ifany")

INCORRECT CORRECT <NA>

515 2619 1686

> table(vocabTest[, myWords[1]], useNA = "ifany")

INCORRECT CORRECT <NA>

515 2619 1686

Thinking of increasing a counter by 1 and repeating several times is called looping, and we will explore

looping more later. For now, we’ll secretly loop via lapply to apply table to the entire dataset. Our goal is

to count all corrects/incorrects at once, rather than doing it piecemeal by typing in the same commands

repeatedly and just changing variable names. Also, while headcounts are nice enough, we generally see such

data summarized via proportions. Let’s work our way backward. At the end, we use do.call to use the rbind

function on the percents of each word correct vs incorrect; do.call simply runs rbind on each percents

value in sequence – more looping! The rbind function is used to simply make it all look pretty (consider

typing in percents into your Rstudio console after running the below code to see why rbind is so helpful).

Before we could do that, we needed to build up percents, which we did by running a proportion table to

create those percents. Since we want a proportion table for each word, we use lapply on our dataset. Of

course, the above tables we had created for just worda were not enough, so we had to create each table, take

a prop.table of their data, store all proportion data into percents, and finally make it all look good as we’ve

done on the next page:

> proportion.table <- function(x) {

+ prop.table( table( x ) )

+ }

>

> percents <- lapply(vocabTest, proportion.table)

>

> do.call(rbind, percents)

INCORRECT CORRECT

worda 0.16432674 0.8356733

wordb 0.06868752 0.9313125

wordc 0.76188761 0.2381124

wordd 0.04441624 0.9555838

worde 0.17356173 0.8264383

wordf 0.18032787 0.8196721

wordg 0.65165877 0.3483412

wordh 0.63088235 0.3691176

wordi 0.23732057 0.7626794

wordj 0.71540984 0.2845902

The GSS dataset also has a variable for the total score on the vocabulary test, which is simply the sum

of the number of words defined correctly. We have added that to the data frame using the cbind function.

I won’t show all the steps here, but will show you after all the recoding of missing data that the distribution

of scores on the vocabulary test is negatively skewed but pretty “normal-looking” by the eyeball test

(see Figure 2-3).