Handling Strings with R

Gaston Sanchez
gastonsanchez.com

Preface

Handling character strings in R? Wait a second . . . you exclaim, *R is not a scripting language like Perl, Python, or Ruby.* Why would you want to use R for handling and processing text? Well, because sooner or later (I would say sooner than later) you will have to deal with some kind of string manipulation for your data analysis. So it's better to be prepared for such tasks and know how to perform them inside the R environment.

If you have been formed and trained in "classical statistics" (as I was), I bet you probably don't think of character strings as data that can be analyzed. The bottom line for your analysis is numbers or things that can be mapped to numeric values. *Text and character strings? Really? Are you kidding me? . . .* That's what I used to think right after finishing college. During my undergraduate studies in statistics, none of my professors mentioned analysis applications with strings and text data. It was years later, in grad school, when I got the chance to be marginally involved with some statistical text analysis.

Perhaps even worse is the not so uncommon believe that string manipulation is a secondary non-relevant task. People will be impressed and will admire you for any kind of fancy model, sophisticated algorithms, and black-box methods that you get to apply. Everybody loves the *haute cuisine* of data analysis and the top notch analytics. But when it comes to processing and manipulating strings, many will think of it as washing the dishes or pealing and cutting potatos. If you want to be perceived as a data chef, you may be tempted to think that you shouldn't waste your time in those boring tasks of manipulating strings. Yes, it is true that you won't get a *Michelin* star for processing character data. But you would hardly become a good data cook if you don't get your hands dirty with string manipulation. And to be

honest, it's not always that boring. Whether you like it or not, no one should ever claim to be a data analyst until he or she has done some manipulation of strings. You don't have to be an expert or some string processing hacker though. But you do need to know the basics and have an idea on how to proceed in case you need to play with text-character-string data.

Text Is Omnipresent

At its heart, computing involves working with numbers. That's the main reason why computers were invented: to facilitate mathematical operations around numbers: from basic arithmetic to more complex operations (e.g. trigonometry, algebra, calculus, etc.) Nowadays, however, computers are used to work with data that are not just numbers. We use them to write a variety of documents, we use them to create and edit images and videos, to manipulate sound, among many other tasks. Learning to manipulate those data types is fundamental to programming.

Just think about it. Today, there is a considerable amount of information and data in the form of text. Look at any website: pretty much the contents are text and images, with some videos here and there, and maybe some tables or list of numbers.

Likewise, most of the times you are going to be working with text files: script files, reports, data files, source code files, etc. All the R script files that you use are essentially plain text files. I bet you have a csv file or any other field delimited format (or even in HTML, XML, JSON, etc), with some fields containing characters. In all of these cases what you are working with is essentially a bunch of characters.

And then inside R we also have text. Things like row names and column names of matrices, data frames, tables, and any other rectangular data structure. Lists may contain names, vectors may also have names. And what about the text in graphics? Things like titles, subtitles, axis labels, legends, colors, displayed text in a plot, etc.

At the end of the day all the data that is passed to the computer is converted to binary format (zeros and ones) so computers can process it. But no one can deny the fact that a lot of what we do with computers is working with text and character strings.

Text is omnipresence. Whether you are aware of this or not, we are surrounded by text.

About This Book

This book aims to help you get started with manipulating strings in R. What type of manipulations am I talking about? For example, I'm sure that you have encountered one or more of the following cases:

- You want to remove a given character in the names of your variables
- You want to replace a given character in your data
- Maybe you wanted to convert labels to upper case (or lower case)
- You've been struggling with xml (or html) files
- You've been modifying text files in excel changing labels, categories, one cell at a time, or doing one thousand copy-paste operations

The content of the book is divided in four major parts:

1. Characters and Strings in R
2. Printing and Formatting
3. Regular Expressions
4. Applications

If you have minimal or none experience with R, the best place to start is Chapter 2: Character Strings in R. If you are already familiar with the basics of vectors and character objects, you can quickly skim this chapter, or skip it, and then go to another chapter of your interest.

Chapters 3 and 4 deal with basic string manipulations. By "basic" I mean any type of manipulation and transformation that does not require the use of regular expressions.

The second part of the book describes different ways to format text and numbers. These are useful tools for when you want to produce output that will either be displayed on screen, or that will be exported to a file.

The third part comprises working with regular expressions (regex). Here you will learn about the basic concepts around regular expressions (regex), the intricacies when working with regex in R, and becoming familiar with the regex functions in the R package `"stringr"`.

Last but not least, the fourth part of the book present a couple of case studies and extended practical examples that cover the main topics covered in the book.

Having said that, I should say that this book is NOT about textual data analysis, linguistic analysis, text mining, or natural language processing.

About The Reader

I am assuming three things about you. In decreasing order of importance:

1. You already know R—this is not an introductory text on R—.
2. You already use R for handling quantitative and qualitative data, but not (necessarily) for processing strings.
3. You have some basic knowledge about Regular Expressions.

Main Resources

I should also say that this work is my third iteration on the subject of manipulating strings, text and character data in R. I started writing the draft of my first manuscript around 2012 when there was not much documentation on how to manipulate character strings in R. Although the number of resources about this subject has increased since then, the pace of these changes has been considerably slow.

What I wrote eight years ago is still valid today. There is not much documentation on how to manipulate character strings and text data in R. There are great R books for an enormous variety of statistical methods, graphics and data visualization, as well as applications in a wide range of fields such as ecology, genetics, psychology, finance, economics, etc. But not much for manipulating strings and text data.

I still believe that the main reason for this lack of resources is that R is not considered to be qualified as a "scripting" language: R is primarily perceived as a language for computing and programming with (mostly numeric) data. Quoting Hadley Wickham (2010)

http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf

> "R provides a solid set of string operations, but because they have grown organically over time, they can be inconsistent and a little hard to learn.

> Additionally, they lag behind the string operations in other programming languages, so that some things that are easy to do in languages like Ruby or Python are rather hard to do in R"

Most introductory books about R have small sections that briefly cover string manipulation without going further down. That is why I don't have many books for recommendation, if anything the book by Phil Spector *Data Manipulation with R*.

If published material is not abundant, we still have the online world. The good news is that the web is full of hundreds of references about processing character strings. The bad news is that they are very spread and uncategorized.

For specific topics and tasks, a good place to start with is *Stack Overflow*. This is a questions-and-answers site for programmers that has a lot of questions related with R. Just look for those questions tagged with `"r"`:

http://stackoverflow.com/questions/tagged/r

There is a good number of posts related with handling characters and text, and they can give you a hint on how to solve a particular problem. There is also *R-bloggers*, http://www.r-bloggers.com, a blog aggregator for R enthusiasts in which is also possible to find contributed material about processing strings as well as text data analysis.

You can also check the following resources that have to do with string manipulations. It is a very short list of resources but I've found them very useful:

- R Wikibook: Programming and Text Processing
  http://en.wikibooks.org/wiki/R_Programming/Text_Processing
  R wikibook has a section dedicated to text processing that is worth check it out.
- stringr: modern, consisting string processing by Hadley Wickham
  http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf
  Article from the R journal introducing the package `stringr` by Hadley Wickham.
- Introduction to String Matching and Modification in R Using Regular Expressions by Svetlana Eden
  http://biostat.mc.vanderbilt.edu/wiki/pub/Main/SvetlanaEdenRFiles/regExprTalk.pdf

For things such as textual data analysis, linguistic analysis, text mining, or natural

language processing, I highly recommend you to take a look at the CRAN Task View on Natural Language Processing (NLP):
http://cran.r-project.org/web/views/NaturalLanguageProcessing.html

While it is true that R may not be as rich and diverse as other scripting languages when it comes to string manipulation, I'm one of those who believe it can take you very far if you know how. By writing this book, my goal is to provide you enough material to do more advanced string and text processing operations. My hope is that, after reading this book, you will have the necessary tools in your toolbox for dealing with these (and many) other situations that involve handling and processing strings in R.

Citation

You can cite this work as:

Sanchez, G. (2021) **Handling Strings with R**
Trowchez Editions. Berkeley, 2021.

Gaston Sanchez
Berkeley, California
December, 2020

Contents

Part I

Characters and Strings in R

Chapter 1

Introductory Appetizer

To give you an idea of some of the things we can do in R with string processing, let's play a bit with a simple example.

1.1 A Toy Example

For this crash informal introduction, we'll use the data frame `USArrests` that already comes with R. Use the function `head()` to get a peek of the data:

```
# take a peek of USArrests
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

The labels on the rows such as `Alabama` or `Alaska` are displayed strings. Likewise, the labels of the columns —`Murder`, `Assault`, `UrbanPop` and `Rape`— are also strings.

1.1.1 Abbreviating strings

Suppose we want to abbreviate the names of the States. Furthermore, suppose we want to abbreviate the names using the first four characters of each name. One way to do that is by using the function `substr()` which substrings a character vector. We just need to indicate the `start=1` and `stop=4` positions:

```
# names of states
states <- rownames(USArrests)

# substr
substr(x = states, start = 1, stop = 4)
```

```
##  [1] "Alab" "Alas" "Ariz" "Arka" "Cali" "Colo" "Conn" "Dela"
##  [9] "Flor" "Geor" "Hawa" "Idah" "Illi" "Indi" "Iowa" "Kans"
## [17] "Kent" "Loui" "Main" "Mary" "Mass" "Mich" "Minn" "Miss"
## [25] "Miss" "Mont" "Nebr" "Neva" "New " "New " "New " "New "
## [33] "Nort" "Nort" "Ohio" "Okla" "Oreg" "Penn" "Rhod" "Sout"
## [41] "Sout" "Tenn" "Texa" "Utah" "Verm" "Virg" "Wash" "West"
## [49] "Wisc" "Wyom"
```

This may not be the best solution. Note that there are four states with the same abbreviation `"New "` (New Hampshire, New Jersey, New Mexico, New York). Likewise, North Carolina and North Dakota share the same name `"Nort"`. In turn, South Carolina and South Dakota got the same abbreviation `"Sout"`.

A better way to abbreviate the names of the states can be performed by using the function `abbreviate()` like so:

```
# abbreviate state names
states2 <- abbreviate(states)

# remove vector names (for convenience)
names(states2) <- NULL
states2
```

```
##  [1] "Albm" "Alsk" "Arzn" "Arkn" "Clfr" "Clrd" "Cnnc" "Dlwr"
##  [9] "Flrd" "Gerg" "Hawa" "Idah" "Illn" "Indn" "Iowa" "Knss"
## [17] "Kntc" "Losn" "Main" "Mryl" "Mssc" "Mchg" "Mnns" "Msss"
## [25] "Mssr" "Mntn" "Nbrs" "Nevd" "NwHm" "NwJr" "NwMx" "NwYr"
```

```
## [33] "NrtC" "NrtD" "Ohio" "Oklh" "Orgn" "Pnns" "RhdI" "SthC"
## [41] "SthD" "Tnns" "Texs" "Utah" "Vrmn" "Vrgn" "Wshn" "WstV"
## [49] "Wscn" "Wymn"
```

If we decide to try an abbreviation with five letters we just simply change the argument `minlength = 5`

```
# abbreviate state names with 5 letters
abbreviate(states, minlength = 5)
```

```
##        Alabama         Alaska        Arizona       Arkansas
##        "Alabm"        "Alask"        "Arizn"        "Arkns"
##     California       Colorado    Connecticut       Delaware
##        "Clfrn"        "Colrd"        "Cnnct"        "Delwr"
##        Florida        Georgia         Hawaii          Idaho
##        "Flord"        "Georg"        "Hawai"        "Idaho"
##       Illinois        Indiana           Iowa         Kansas
##        "Illns"        "Indin"         "Iowa"        "Kanss"
##       Kentucky      Louisiana          Maine       Maryland
##        "Kntck"        "Lousn"        "Maine"        "Mryln"
##  Massachusetts       Michigan     Minnesota    Mississippi
##        "Mssch"        "Mchgn"        "Mnnst"        "Mssss"
##       Missouri        Montana       Nebraska         Nevada
##        "Missr"        "Montn"        "Nbrsk"        "Nevad"
##  New Hampshire     New Jersey     New Mexico       New York
##        "NwHmp"        "NwJrs"        "NwMxc"        "NwYrk"
## North Carolina   North Dakota           Ohio       Oklahoma
##        "NrthC"        "NrthD"         "Ohio"        "Oklhm"
##         Oregon   Pennsylvania   Rhode Island South Carolina
##        "Oregn"        "Pnnsy"        "RhdIs"        "SthCr"
##   South Dakota      Tennessee          Texas           Utah
##        "SthDk"        "Tnnss"        "Texas"         "Utah"
##        Vermont       Virginia     Washington  West Virginia
##        "Vrmnt"        "Virgn"        "Wshng"        "WstVr"
##      Wisconsin        Wyoming
##        "Wscns"        "Wymng"
```

1.1.2 Getting the longest name

Now let's imagine that we need to find the longest name. This implies that we need to count the number of letters in each name. The function `nchar()` comes handy for that purpose. Here's how we could do it:

```
# size (in characters) of each name
state_chars = nchar(states)
state_chars
```

```
##  [1]  7  6  7  8 10  8 11  8  7  7  6  5  8  7  4  6  8  9
## [19]  5  8 13  8  9 11  8  7  8  6 13 10 10  8 14 12  4  8
## [37]  6 12 12 14 12  9  5  4  7  8 10 13  9  7
```

```
# longest name
states[which(state_chars == max(state_chars))]
```

```
## [1] "North Carolina" "South Carolina"
```

1.1.3 Selecting States

To make things more interesting, let's assume that we wish to select those states containing the letter `"k"`. How can we do that? Very simple, we just need to use the function `grep()` for working with regular expressions. Simply indicate the `pattern = "k"` as follows:

```
# get states names with 'k'
grep(pattern = "k", x = states, value = TRUE)
```

```
## [1] "Alaska"       "Arkansas"     "Kentucky"
## [4] "Nebraska"     "New York"     "North Dakota"
## [7] "Oklahoma"     "South Dakota"
```

Instead of grabbing those names containing `"k"`, say we wish to select those states containing the letter `"w"`. Again, this can be done with `grep()`:

```
# get states names with 'w'
grep(pattern = "w", x = states, value = TRUE)
```

```
## [1] "Delaware"      "Hawaii"        "Iowa"
## [4] "New Hampshire" "New Jersey"    "New Mexico"
## [7] "New York"
```

Notice that we only selected those states with lowercase "w". But what about those states with uppercase "W"? There are several options to find a solution for this question. One option is to specify the searched pattern as a character class "[wW]":

```
# get states names with 'w' or 'W'
grep(pattern = "[wW]", x = states, value = TRUE)
```

```
##  [1] "Delaware"      "Hawaii"        "Iowa"
##  [4] "New Hampshire" "New Jersey"    "New Mexico"
##  [7] "New York"      "Washington"    "West Virginia"
## [10] "Wisconsin"     "Wyoming"
```

Another solution is to first convert the state names to lower case, and then look for the character "w", like so:

```
# get states names with 'w'
grep(pattern = "w", x = tolower(states), value = TRUE)
```

```
##  [1] "delaware"      "hawaii"        "iowa"
##  [4] "new hampshire" "new jersey"    "new mexico"
##  [7] "new york"      "washington"    "west virginia"
## [10] "wisconsin"     "wyoming"
```

Alternatively, instead of converting the state names to lower case we could do the opposite (convert to upper case), and then look for the character "W", like so:

```
# get states names with 'W'
grep(pattern = "W", x = toupper(states), value = TRUE)
```

```
##  [1] "DELAWARE"      "HAWAII"        "IOWA"
##  [4] "NEW HAMPSHIRE" "NEW JERSEY"    "NEW MEXICO"
##  [7] "NEW YORK"      "WASHINGTON"    "WEST VIRGINIA"
## [10] "WISCONSIN"     "WYOMING"
```

A third solution involves specifying the argument `ignore.case=TRUE` inside `grep()`:

```
# get states names with 'w'
grep(pattern = "w", x = states, value = TRUE, ignore.case = TRUE)
```

```
##  [1] "Delaware"      "Hawaii"        "Iowa"
##  [4] "New Hampshire" "New Jersey"    "New Mexico"
##  [7] "New York"      "Washington"    "West Virginia"
```

```
## [10] "Wisconsin"     "Wyoming"
```

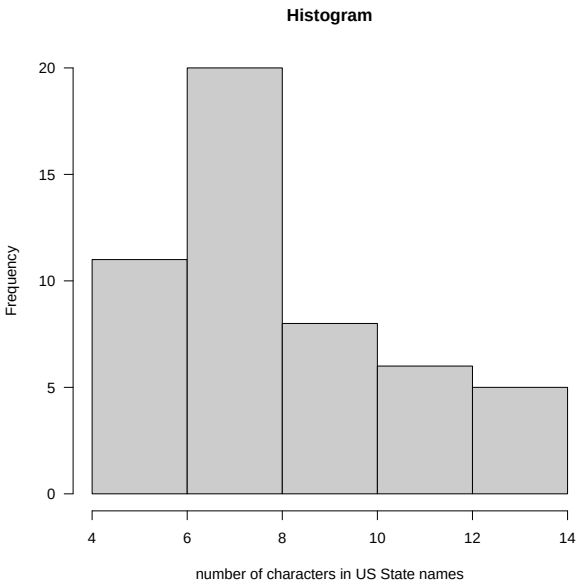
1.1.4 Some computations

Besides manipulating strings and performing pattern matching operations, we can also do some computations. For instance, we could ask for the distribution of the State names' length. To find the answer we can use `nchar()`. Furthermore, we can plot a histogram of such distribution:

```
summary(nchar(states))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    4.00    7.00    8.00    8.44   10.00   14.00

# histogram
hist(nchar(states), las = 1, col = "gray80", main = "Histogram",
     xlab = "number of characters in US State names")
```



Let's ask a more interesting question. What is the distribution of the vowels in the names of the States? For instance, let's start with the number of `a`'s in each name. There's a very useful function for this purpose: `regexpr()`. We can use `regexpr()` to get the number of times that a searched pattern is found in a character vector. When there is no match, we get a value -1.

```
# position of a's
positions_a <- gregexpr(pattern="a", text=states, ignore.case = TRUE)

# how many a's?
num_a <- sapply(positions_a, function(x) ifelse(x[1]>0, length(x), 0))
num_a
```

```
##  [1] 4 3 2 3 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2
## [29] 1 0 0 0 2 2 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

If you inspect `positions_a` you'll see that it contains some negative numbers `-1`. This means there are no letters `a` in that name. To get the number of occurrences of `a`'s we are taking a shortcut with `sapply()`.

The same operation can be performed by using the function `str_count()` from the package `"stringr"`.

```
# load stringr (remember to install it first)
library(stringr)

# total number of a's
str_count(states, "a")
```

```
##  [1] 3 2 1 2 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2
## [29] 1 0 0 0 2 2 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

Notice that we are only getting the number of `a`'s in lower case. Since `str_count()` does not contain the argument `ignore.case`, we need to transform all letters to lower case, and then count the number of `a`'s like this:

```
# total number of a's
str_count(tolower(states), "a")
```

```
##  [1] 4 3 2 3 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2
## [29] 1 0 0 0 2 2 0 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

Once we know how to do it for one vowel, we can do the same for all the vowels:

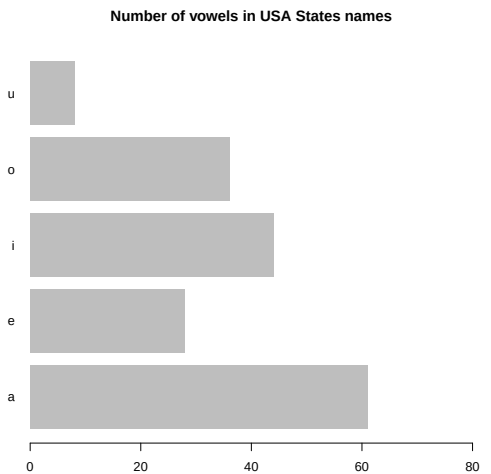
```
# calculate number of vowels in each name
vowels <- c("a", "e", "i", "o", "u")
num_vowels <- vector(mode = "integer", length = 5)

for (j in seq_along(vowels)) {
  num_aux <- str_count(tolower(states), vowels[j])
  num_vowels[j] <- sum(num_aux)
}

# sort them in decreasing order
names(num_vowels) <- vowels
sort(num_vowels, decreasing = TRUE)
```

```
##  a  i  o  e  u
## 61 44 36 28  8
```

```
# barplot
barplot(num_vowels, main = "Number of vowels in USA States names",
        border = NA, xlim = c(0, 80), las = 1, horiz = TRUE)
```



Chapter 2

Character Strings in R

2.1 Introduction

This chapter introduces you to the basic concepts for creating character vectors and character strings in R. You will also learn how R treats objects containing characters.

2.2 Characters in R

In R, a piece of text is represented as a sequence of characters (letters, numbers, and symbols). The data type that R provides for storing sequences of characters is *character*. Formally, the class of object that holds character strings in R is `"character"`.

We express character strings by surrounding text within single quotes:

```
'a character string using single quotes'
```

or we can also surround text within double quotes:

```
"a character string using double quotes"
```

The important thing is that we must match the type of quotes that we are using. A starting single quote must have an ending single quote. Likewise, a string with

an opening double quote must be closed with a double quote.

Typing characters in R like in above examples is not very useful. Typically, we are going to create objects or variables containing some strings. For example, we can create a variable `string` that stores some string:

```
string <- 'do more with less'
string

## [1] "do more with less"
```

Notice that when you print a character object, R displays it using double quotes (regardless of whether the string was created using single or double quotes). This allows us to quickly identify when an object contains character strings.

When writing strings, we can insert single quotes in a string with double quotes, and vice versa:

```
"The 'R' project for statistical computing"
```

```
'The "R" project for statistical computing'
```

However, we cannot directly insert single quotes in a string with single quotes, neither we can insert double quotes in a string with double quotes (Don't do this!):

```
"This "is" totally unacceptable"
```

```
'This 'is' absolutely wrong'
```

In both cases R will give you an error due to the unexpected presence of either a double quote within double quotes, or a single quote within single quotes.

If we really want to include a double quote as part of the string, we need to *escape* the double quote using a backslash "\" before it:

```
"The \"R\" project for statistical computing"
```

We will talk more about escaping characters in the following chapters.

2.3 Getting Started with Strings

Perhaps the most common use of character strings in R have to do with:

- names of files
- names of data objects
- text displayed in plots

When we read a file, for instance a data table stored in a csv file, we typiclaly use the `read.table()` function. Assuming that the file is in the working directory:

```
dat <- read.csv(file = 'dataset.csv')
```

The main parameter for the function `read.csv()` is `file` which requires a character string with the pathname of the file.

Another example of a basic use of characters is when we assign names to the elements of some data structure in R. For instance, if you want to name the elements of a (numeric) vector, you can use the function `names()` as follows:

```
num_vec <- 1:5
names(num_vec) <- c('uno', 'dos', 'tres', 'cuatro', 'cinco')
num_vec
```

Likewise, many of the parameters in the plotting functions require some sort of input string. Below is a hypothetical example of a scatterplot that includes several graphical elements like the main title (`main`), subtitle (`sub`), labels for both x-axis and y-axis (`xlab, ylab`), the name of the color, and the symbol for the point character (`pch`)

```
plot(x, y,
     main = 'Main Title',
     sub = 'Subtitle',
     xlab = 'x-axis label',
     ylab = 'y-axis label',
     col = 'red',
     pch = 'x')
```

2.4 Creating Character Strings

Besides the single quotes `''` or double quotes `""`, R provides the function `character()` to create character strings. More specifically, `character()` is the function that creates vector objects of type `"character"`.

When using `character()` you just have to specify the length of the vector. The output will be a character vector filled of empty strings:

```
# character vector with 5 empty strings
char_vector <- character(5)
char_vector

## [1] "" "" "" "" ""
```

When would you use `character()`? A typical usage case is when you want to initialize an empty character vector of a given length. The idea is to create an object that you will modify later with some computation.

As with any other vector, once an empty character vector has been created, you can add new components to it by simply giving it an index value outside its previous range:

```
# another example
example <- character(0)
example

## character(0)

# check its length
length(example)

## [1] 0

# add first element
example[1] <- "first"
example

## [1] "first"

# check its length again
length(example)

## [1] 1
```

You can add more elements without the need to follow a consecutive index range:

```
example[4] <- "fourth"
example
```

```
## [1] "first"  NA       NA       "fourth"

length(example)

## [1] 4
```

Notice that the vector `example` went from containing one-element to contain four-elements without specifying the second and third elements. R fills this gap with missing values `NA`.

2.4.1 Empty string

The most basic string: the **empty string** produced by consecutive quotation marks: `""`. Technically, `""` is a string with no characters in it, hence the name *empty string*:

```
# empty string
empty_str <- ""

# display
empty_str

## [1] ""

# class
class(empty_str)

## [1] "character"
```

2.4.2 Empty character vector

Another basic string structure is the **empty character vector** produced by the function `character()` and its argument `length=0`:

```
# empty character vector
empty_chr <- character(0)

# display
empty_chr

## character(0)

# class
```

```
class(empty_chr)
```

```
## [1] "character"
```

It is important not to confuse the empty character vector `character(0)` with the empty string `""`; one of the main differences between them is that they have different lengths:

```
# length of empty string
length(empty_str)
```

```
## [1] 1
```

```
# length of empty character vector
length(empty_chr)
```

```
## [1] 0
```

Notice that the empty string `empty_str` has length 1, while the empty character vector `empty_chr` has length 0.

Also, `character(0)` occurs when we have a character vector with one or more elements, and we attempt to subset the position 0:

```
string <- c('sun', 'sky', 'clouds')
string
```

```
## [1] "sun"    "sky"    "clouds"
```

If we try to retrieve the element in position 0 we get:

```
string[0]
```

```
## character(0)
```

2.4.3 Function `c()`

There is also the generic function `c()` (concatenate or combine) that we can use to create character vectors. Simply pass any number of elements separated by commas:

```
string <- c('sun', 'sky', 'clouds')
string
```

```
## [1] "sun"    "sky"    "clouds"
```

Again, notice that we can use single or double quotes to define the character elements inside `c()`

```
planets <- c("mercury", 'venus', "mars")
planets
```

```
## [1] "mercury" "venus"   "mars"
```

2.4.4 `is.character()` and `as.character()`

Related to `character()` we have its two sister functions: `as.character()` and `is.character()`. These two functions are generic methos for creating objects of type `"character"` and testing whether an R object is of type `"character"`. For instance, let's define two objects `a` and `b` as follows:

```
# define two objects 'a' and 'b'
a <- "test me"
b <- 8 + 9
```

To test if `a` and `b` are of type `"character"` use the function `is.character()`:

```
# are 'a' and 'b' characters?
is.character(a)
```

```
## [1] TRUE
```

```
is.character(b)
```

```
## [1] FALSE
```

Likewise, you can also use the function `class()` to get the class of an object:

```
# classes of 'a' and 'b'
class(a)
```

```
## [1] "character"
```

```
class(b)
```

```
## [1] "numeric"
```

The function `as.character()` is a coercing method. For better or worse, R allows

you to convert (i.e. coerce) non-character objects into character strings with the function `as.character()`:

```
# converting 'b' as character
b <- as.character(b)
b

## [1] "17"
```

2.5 Strings and R Objects

Before continuing our discussion on functions for manipulating strings, we need to talk about some important technicalities. R has five main types of objects to store data: `vector`, `factor`, `matrix` (and `array`), `data.frame`, and `list`. We can use each of those objects to store character strings. However, these objects will behave differently depending on whether we store `character` data with other types of data. Let's see how R treats objects with different types of data (e.g. character, numeric, logical).

2.5.1 Behavior of R objects with character strings

Vectors The most basic type of data container are vectors. You can think of vectors as the building blocks for other more complex data structures. R has six types of vectors, technically referred to as *atomic types* or atomic vectors: logical, integer, double, character, complex, and raw.

Types of R vectors

Type	Description
`logical`	a vector containing logical values
`integer`	a vector containing integer values
`double`	a vector containing real values
`character`	a vector containing character values
`complex`	a vector containing complex values
`raw`	a vector containing bytes

Vectors are atomic structures because their values must be *all of the same mode*. This means that any given vector must be unambiguously either logical, numeric,

complex, character or raw.

So what happens when we mix different types of data in a vector?

```
# vector with numbers and characters
c(1:5, pi, "text")
```

```
## [1] "1"                "2"                "3"
## [4] "4"                "5"                "3.14159265358979"
## [7] "text"
```

As you can tell, the resulting vector from combining integers (`1:5`), the number `pi`, and some `"text"` is a vector with all its elements treated as character strings. In other words, when we combine mixed data in vectors, strings will dominate. This means that the mode of the vector will be `"character"`, even if we mix logical values:

```
# vector with numbers, logicals, and characters
c(1:5, TRUE, pi, "text", FALSE)
```

```
## [1] "1"                "2"                "3"
## [4] "4"                "5"                "TRUE"
## [7] "3.14159265358979" "text"             "FALSE"
```

In fact, R follows two basic rules of data types coercion. The most strict rule is: if a character string is present in a vector, everything else in the vector will be converted to character strings. The other coercing rule is: if a vector only has logicals and numbers, then logicals will be converted to numbers; `TRUE` values become 1, and `FALSE` values become 0.

Keeping these rules in mind will save you from many headaches and frustrating moments. Moreover, you can use them in your favor to manipulate data in very useful ways.

Matrices The same behavior of vectors happens when we mix characters and numbers in matrices. Again, everything will be treated as characters:

```
# matrix with numbers and characters
rbind(1:5, letters[1:5])
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "1"  "2"  "3"  "4"  "5"
```

```
## [2,] "a"  "b"  "c"  "d"  "e"
```

Data frames With data frames, things are a bit different. By default, character strings inside a data frame will be converted to factors:

```
# data frame with numbers and characters
df1 = data.frame(numbers=1:5, letters=letters[1:5])
df1
```

```
##   numbers letters
## 1       1       a
## 2       2       b
## 3       3       c
## 4       4       d
## 5       5       e
```

```
# examine the data frame structure
str(df1)
```

```
## 'data.frame': 5 obs. of  2 variables:
##  $ numbers: int  1 2 3 4 5
##  $ letters: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
```

To turn-off the `data.frame()`'s default behavior of converting strings into factors, use the argument `stringsAsFactors = FALSE`:

```
# data frame with numbers and characters
df2 <- data.frame(
  numbers = 1:5,
  letters = letters[1:5],
  stringsAsFactors = FALSE)

df2
```

```
##   numbers letters
## 1       1       a
## 2       2       b
## 3       3       c
## 4       4       d
## 5       5       e
```

```
# examine the data frame structure
str(df2)
```

```
## 'data.frame': 5 obs. of  2 variables:
##  $ numbers: int  1 2 3 4 5
##  $ letters: chr  "a" "b" "c" "d" ...
```

Even though `df1` and `df2` are identically displayed, their structure is different. While `df1$letters` is stored as a `"factor"`, `df2$letters` is stored as a `"character"`.

Lists With lists, we can combine whatever type of data we want. The type of data in each element of the list will maintain its corresponding mode:

```
# list with elements of different mode
list(1:5, letters[1:5], rnorm(5))
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] "a" "b" "c" "d" "e"
##
## [[3]]
## [1] -0.5375337 -0.4160729 -0.9606883 -0.9394698  0.8592440
```

2.6 The Workhorse Function `paste()`

The function `paste()` is perhaps one of the most important functions that we can use to create and build strings. `paste()` takes one or more R objects, converts them to `"character"`, and then it concatenates (pastes) them to form one or several character strings. Its usage has the following form:

```
paste(..., sep = " ", collapse = NULL)
```

The argument `...` means that it takes any number of objects. The argument `sep` is a character string that is used as a separator. The argument `collapse` is an optional string to indicate if we want all the terms to be collapsed into a single string. Here is a simple example with `paste()`:

```
# paste
PI = paste("The life of", pi)

PI

## [1] "The life of 3.14159265358979"
```

As you can see, the default separator is a blank space (`sep = " "`). But you can select another character, for example `sep = "-"`:

```
# paste
IloveR = paste("I", "love", "R", sep = "-")

IloveR

## [1] "I-love-R"
```

If we give `paste()` objects of different length, then it will apply a recycling rule. For example, if we paste a single character `"X"` with the sequence `1:5`, and separator `sep = "."` this is what we get:

```
# paste with objects of different lengths
paste("X", 1:5, sep = ".")

## [1] "X.1" "X.2" "X.3" "X.4" "X.5"
```

To see the effect of the `collapse` argument, let's compare the difference with collapsing and without it:

```
# paste with collapsing
paste(1:3, c("!","?","+"), sep = '', collapse = "")

## [1] "1!2?3+"

# paste without collapsing
paste(1:3, c("!","?","+"), sep = '')

## [1] "1!" "2?" "3+"
```

One of the potential problems with `paste()` is that it coerces missing values `NA` into the character `"NA"`:

```
# with missing values NA
evalue = paste("the value of 'e' is", exp(1), NA)

evalue
```

```
## [1] "the value of 'e' is 2.71828182845905 NA"
```

In addition to `paste()`, there's also the function `paste0()` which is the equivalent of

```
 paste(..., sep = "", collapse)
```

```
# collapsing with paste0
paste0("let's", "collapse", "all", "these", "words")
```

```
## [1] "let'scollapseallthesewords"
```

2.7 Getting Text into R

We've seen how to express character strings using single quotes `''` or double quotes `""`. But we also need to discuss how to get text into R, that is, how to import and read files that contain character strings. So, how do we get text into R? Well, it basically depends on the type-format of the files we want to read.

We will describe two general situations. One in which the content of the file can be represented in tabular format (i.e. rows and columns). The other one when the content does not have a tabular structure. In this second case, we have characters that are in an unstructured form (i.e. just lines of strings) or at least in a non-tabular format such as html, xml, or other markup language format.

Another function is `scan()` which allows us to read data in several formats. Usually we use `scan()` to parse R scripts, but we can also use to import text (characters)

2.7.1 Reading tables

If the data we want to import is in some tabular format (i.e. cells and columns) we can use the set of functions to read tables like `read.table()` and its sister functions—e.g. `read.csv()`, `read.delim()`, `read.fwf()`—. These functions read

a file in table format and create a data frame from it, with rows corresponding to cases, and columns corresponding to fields in the file.

Functions to read files in tabular format

Function	Description
`read.table()`	main function to read file in table format
`read.csv()`	reads csv files separated by a comma `","`
`read.csv2()`	reads csv files separated by a semicolon `";"`
`read.delim()`	reads files separated by tabs `"\t"`
`read.delim2()`	similar to `read.delim()`
`read.fwf()`	read fixed width format files

A word of caution about the built-in functions to read data tables: by default they all convert characters into R factors. This means that if there is a column with characters, R will treat this data as qualitative variable. To turn off this behavior, we need to specify the argument `stringsAsFactors = FALSE`. In this way, all the characters in the imported file will be kept as characters once we read them in R.

Let's see a simple example reading a file from the Australian radio broadcaster *ABC* (http://www.abc.net.au/radio/). In particular, we'll read a `csv` file that contains data from ABC's radio stations. Such file is located at:
http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv

To import the file `abc-local-radio.csv`, we can use either `read.table()` or `read.csv()` (just choose the right parameters). Here's the code to read the file with `read.table()`:

```
# assembling url
abc <- "http://www.abc.net.au/"
radios <- "local/data/public/stations/abc-local-radio.csv"
abc_radiosl <- paste0(abc, radios)

# read data from URL
radio <- read.table(
  file = abc_radios,
  header = TRUE,
  sep = ',',
  stringsAsFactors = FALSE)
```

In this case, the location of the file is defined in the object `abc` which is the first argument passed to `read.table()`. Then we choose other arguments such as `header = TRUE`, `sep = ","`, and `stringsAsFactors = FALSE`. The argument `header = TRUE` indicates that the first row of the file contains the names of the columns. The separator (a comma) is specifcied by `sep = ","`. And finally, to keep the character strings in the file as `"character"` in the data frame, we use `stringsAsFactors = FALSE`.

If everything went fine during the file reading operation, the next thing to do is to chek the size of the created data frame using `dim()`:

```
# size of table in 'radio'
dim(radio)

## [1] 53 18
```

Notice that the data frame `radio` is a table with 53 rows and 18 columns. If we examine ths structure with `str()` we will get information of each column. The argument `vec.len = 1` indicates that we just want the first element of each variable to be displayed:

```
# structure of columns
str(radio, vec.len = 1)

## 'data.frame': 53 obs. of  18 variables:
##  $ State           : chr  "QLD" ...
##  $ Website.URL     : chr  "http://www.abc.net.au/brisbane/" ...
##  $ Station         : chr  "ABC Radio Brisbane" ...
##  $ Town            : chr  " Brisbane " ...
##  $ Latitude        : num  -27.5 ...
##  $ Longitude       : num  153 ...
##  $ Talkback.number : chr  "1300 222 612" ...
##  $ Enquiries.number: chr  "07 3377 5222" ...
##  $ Fax.number      : chr  "07 3377 5612" ...
##  $ Sms.number      : chr  "0467 922 612" ...
##  $ Street.number   : chr  "114 Grey Street" ...
##  $ Street.suburb   : chr  "South Brisbane" ...
##  $ Street.postcode : int  4101 4700 ...
##  $ PO.box          : chr  "GPO Box 9994" ...
##  $ PO.suburb       : chr  "Brisbane" ...
```

```
##  $ PO.postcode     : int  4001 4700 ...
##  $ Twitter         : chr  " abcbrisbane" ...
##  $ Facebook        : chr  " https://www.facebook.com/abcinbrisbane" ...
```

As you can tell, most of the 18 variables are in `"character"` mode. Only `$Latitude`, `$Longitude`, `$Street.postcode` and `$PO.postcode` have a different mode.

2.7.2 Reading raw text

If what we want is to import text *as is* (i.e. we want to read raw text) then we need to use the function `readLines()`. This function is the one we should use if we don't want R to assume that the data is in any particular form.

The way we work with `readLines()` is by passing it the name of a file or the name of a URL that we want to read. The output is a character vector with one element for each line of the file or url. The produced vector will contain as many elements as lines in the read file.

Let's see how to read a text file. For this example we will use a text file from the site *TEXTFILES.COM* (by Jason Scott) http://www.textfiles.com/music/ . This site contains a section of music related text files. For demonstration purposes let's consider the "Top 105.3 songs of 1991" according to the "Modern Rock" radio station *KITS San Francisco*. The corresponding `txt` file is located at: http://www.textfiles.com/music/ktop100.txt.

To read the file with the function `readLines()`:

```
# read 'ktop100.txt' file
top105 <- readLines("http://www.textfiles.com/music/ktop100.txt")
```

`readLines()` creates a character vector in which each element represents the lines of the URL we are trying to read. To know how many elements (i.e how many lines) are in `top105` we can use the function `length()`. To inspect the first elements (i.e. first lines of the text file) use `head()`

```
# how many lines
length(top105)

## [1] 123

# inspecting first elements
```

```
head(top105)
```

```
## [1] "From: ed@wente.llnl.gov (Ed Suranyi)"
## [2] "Date: 12 Jan 92 21:23:55 GMT"
## [3] "Newsgroups: rec.music.misc"
## [4] "Subject: KITS' year end countdown"
## [5] ""
## [6] ""
```

Looking at the output provided by `head()` the first four lines contain some information about the subject of the email (KITS' year end countdown). The fifth and sixth lines are empty lines. If we inspect the next few lines, we'll see that the list of songs in the top100 starts at line number 11.

```
# top 5 songs
top105[11:15]
```

```
## [1] "1. NIRVANA                      SMELLS LIKE TEEN SPIRIT"
## [2] "2. EMF                          UNBELIEVABLE"
## [3] "3. R.E.M.                       LOSING MY RELIGION"
## [4] "4. SIOUXSIE & THE BANSHEES      KISS THEM FOR ME"
## [5] "5. B.A.D. II                    RUSH"
```

Each line has the ranking number, followed by a dot, followed by a blank space, then the name of the artist/group, followed by a bunch of white spaces, and then the title of the song. As you can see, the number one hit of 1991 was "Smells like teen spirit" by *Nirvana*.

What about the last songs in KITS' ranking? In order to get the answer we can use the `tail()` function to inspect the last `n = 10` elements of the file:

```
# inspecting last 10 elements
tail(top105, n = 10)
```

```
##   [1] "101. SMASHING PUMPKINS          SIVA"
##   [2] "102. ELVIS COSTELLO             OTHER SIDE OF ..."
##   [3] "103. SEERS                      PSYCHE OUT"
##   [4] "104. THRILL KILL CULT           SEX ON WHEELZ"
##   [5] "105. MATTHEW SWEET              I'VE BEEN WAITING"
##   [6] "105.3  LATOUR                   PEOPLE ARE STILL HAVING SEX"
```

```
##  [7] ""
##  [8] "Ed"
##  [9] "ed@wente.llnl.gov"
## [10] ""
```

Note that the last four lines don't contain information about the songs. Moreover, the number of songs does not stop at 105. In fact the ranking goes till 106 songs (last number being 105.3)

We'll stop here the discussion of this chapter. However, it is importat to keep in mind that text files come in a great variety of forms, shapes, sizes, and flavors. For more information on how to import files in R, the authoritative document is the guide on **R Data Import/Export** (by the R Core Team) available at:
http://cran.r-project.org/doc/manuals/r-release/R-data.html

Chapter 3

Basic Manipulations With "base" Functions

3.1 Introduction

In this chapter you will learn about the different functions to do what I call basic manipulations. By "basic" I mean transforming and processing strings in such way that we do not require to use regular expressions. More advanced manipulations involve defining patterns of text and matching such patterns. This is the essential idea behind regular expressions, which is the content of part 3 in this book.

3.2 Basic String Manipulations

Besides creating and printing strings, there are a number of very handy functions in R for doing some basic manipulation of strings. In this section we will review the following functions:

Manipulation of strings

Function	Description
`nchar()`	number of characters
`tolower()`	convert to lower case
`toupper()`	convert to upper case
`casefold()`	case folding
`chartr()`	character translation
`abbreviate()`	abbreviation
`substring()`	substrings of a character vector
`substr()`	substrings of a character vector

3.2.1 Count number of characters with `nchar()`

One of the main functions for manipulating character strings is `nchar()` which counts the number of characters in a string. In other words, `nchar()` provides the "length" of a string:

```
# how many characters?
nchar(c("How", "many", "characters?"))

## [1]  3  4 11

# how many characters?
nchar("How many characters?")

## [1] 20
```

Notice that the white spaces between words in the second example are also counted as characters.

It is important not to confuse `nchar()` with `length()`. While the former gives us the **number of characters**, the later only gives the **number of elements** in a vector.

```
# how many elements?
length(c("How", "many", "characters?"))

## [1] 3

# how many elements?
length("How many characters?")
```

```
## [1] 1
```

3.2.2 Convert to lower case with `tolower()`

R comes with three functions for text casefolding. The first function we'll discuss is `tolower()` which converts any upper case characters into lower case:

```
# to lower case
tolower(c("aLL ChaRacterS in LoweR caSe", "ABCDE"))

## [1] "all characters in lower case"
## [2] "abcde"
```

3.2.3 Convert to upper case with `toupper()`

The opposite function of `tolower()` is `toupper`. As you may guess, this function converts any lower case characters into upper case:

```
# to upper case
toupper(c("All ChaRacterS in Upper Case", "abcde"))

## [1] "ALL CHARACTERS IN UPPER CASE"
## [2] "ABCDE"
```

3.2.4 Upper or lower case conversion with `casefold()`

The third function for case-folding is `casefold()` which is a wraper for both `tolower()` and `toupper()`. Its uasge has the following form:

```
casefold(x, upper = FALSE)
```

By default, `casefold()` converts all characters to lower case, but we can use the argument `upper = TRUE` to indicate the opposite (i.e. characters in upper case):

```
# lower case folding
casefold("aLL ChaRacterS in LoweR caSe")

## [1] "all characters in lower case"

# upper case folding
casefold("All ChaRacterS in Upper Case", upper = TRUE)
```

```
## [1] "ALL CHARACTERS IN UPPER CASE"
```

3.2.5 Character translation with chartr()

There's also the function chartr() which stands for *character translation*. chartr() takes three arguments: an old string, a new string, and a character vector x:

```
chartr(old, new, x)
```

The way chartr() works is by replacing the characters in old that appear in x by those indicated in new. For example, suppose we want to translate the letter a (lower case) with A (upper case) in the sentence x:

```
# replace 'a' by 'A'
chartr("a", "A", "This is a boring string")

## [1] "This is A boring string"
```

It is important to note that old and new must have the same number of characters, otherwise you will get a nasty error message like this one:

```
# incorrect use
chartr("ai", "X", "This is a bad example")

## Error in chartr("ai", "X", "This is a bad example"):  'old' is longer
than 'new'
```

Here's a more interesting example with old = "aei" and new = "#!?". This implies that any a in x will be replaced by #, any e in x will be replaced by ?, and any i in x will be replaced by ?:

```
# multiple replacements
crazy = c("Here's to the crazy ones", "The misfits", "The rebels")
chartr("aei", "#!?", crazy)

## [1] "H!r!'s to th! cr#zy on!s" "Th! m?sf?ts"
## [3] "Th! r!b!ls"
```

3.2.6 Abbreviate strings with abbreviate()

Another useful function for basic manipulation of character strings is abbreviate(). Its usage has the following structure:

```
abbreviate(names.org, minlength = 4, dot = FALSE, strict = FALSE,
           method = c("left.keep", "both.sides"))
```

Although there are several arguments, the main parameter is the character vector (`names.org`) which will contain the names that we want to abbreviate:

```
# some color names
some_colors = colors()[1:4]
some_colors

## [1] "white"         "aliceblue"     "antiquewhite"
## [4] "antiquewhite1"

# abbreviate (default usage)
colors1 = abbreviate(some_colors)
colors1

##         white     aliceblue  antiquewhite antiquewhite1
##        "whit"        "alcb"        "antq"        "ant1"

# abbreviate with 'minlength'
colors2 = abbreviate(some_colors, minlength=5)
colors2

##         white     aliceblue  antiquewhite antiquewhite1
##       "white"       "alcbl"       "antqw"       "antq1"

# abbreviate
colors3 = abbreviate(some_colors, minlength=3, method="both.sides")
colors3

##         white     aliceblue  antiquewhite antiquewhite1
##         "wht"         "alc"         "ant"         "an1"
```

3.2.7 Replace substrings with `substr()`

One common operation when working with strings is the extraction and replacement of some characters. For such tasks we have the function `substr()` which extracts or replaces substrings in a character vector. Its usage has the following form:

```
substr(x, start, stop)
```

`x` is a character vector, `start` indicates the first element to be replaced, and `stop`

indicates the last element to be replaced:

```
# extract 'bcd'
substr("abcdef", 2, 4)

## [1] "bcd"

# replace 2nd letter with hash symbol
x = c("may", "the", "force", "be", "with", "you")
substr(x, 2, 2) <- "#"
x

## [1] "m#y"   "t#e"   "f#rce" "b#"    "w#th"  "y#u"

# replace 2nd and 3rd letters with happy face
y = c("may", "the", "force", "be", "with", "you")
substr(y, 2, 3) <- ":)"
y

## [1] "m:)"   "t:)"   "f:)ce" "b:"    "w:)h"  "y:)"

# replacement with recycling
z = c("may", "the", "force", "be", "with", "you")
substr(z, 2, 3) <- c("#", "@")
z

## [1] "m#y"   "t@e"   "f#rce" "b@"    "w#th"  "y@u"
```

3.2.8 Replace substrings with `substring()`

Closely related to `substr()`, the function `substring()` extracts or replaces substrings in a character vector. Its usage has the following form:

```
substring(text, first, last = 1000000L)
```

`text` is a character vector, `first` indicates the first element to be replaced, and `last` indicates the last element to be replaced:

```
# same as 'substr'
substring("ABCDEF", 2, 4)

## [1] "BCD"
```

```
substr("ABCDEF", 2, 4)

## [1] "BCD"

# extract each letter
substring("ABCDEF", 1:6, 1:6)

## [1] "A" "B" "C" "D" "E" "F"

# multiple replacement with recycling
text = c("more", "emotions", "are", "better", "than", "less")
substring(text, 1:3) <- c(" ", "zzz")
text

## [1] " ore"     "ezzzions" "ar "      "zzzter"   "t an"
## [6] "lezz"
```

3.3 Set Operations

R has dedicated functions for performing set operations on two given vectors. This implies that we can apply functions such as set union, intersection, difference, equality and membership, on "character" vectors.

Set Operations

Function	Description
union()	set union
intersect()	intersection
setdiff()	set difference
setequal()	equal sets
identical()	exact equality
is.element()	is element
%in%()	contains
sort()	sorting
paste(rep())	repetition

3.3.1 Set union with union()

Let's start our reviewing of set functions with union(). As its name indicates, we can use union() when we want to obtain the elements of the union between two

character vectors:

```
# two character vectors
set1 = c("some", "random", "words", "some")
set2 = c("some", "many", "none", "few")

# union of set1 and set2
union(set1, set2)
```

```
## [1] "some"   "random" "words"  "many"   "none"   "few"
```

Notice that `union()` discards any duplicated values in the provided vectors. In the previous example the word `"some"` appears twice inside `set1` but it appears only once in the union. In fact all the set operation functions will discard any duplicated values.

3.3.2 Set intersection with `intersect()`

Set intersection is performed with the function `intersect()`. We can use this function when we wish to get those elements that are common to both vectors:

```
# two character vectors
set3 = c("some", "random", "few", "words")
set4 = c("some", "many", "none", "few")

# intersect of set3 and set4
intersect(set3, set4)
```

```
## [1] "some" "few"
```

3.3.3 Set difference with `setdiff()`

Related to the intersection, we might be interested in getting the difference of the elements between two character vectors. This can be done with `setdiff()`:

```
# two character vectors
set5 = c("some", "random", "few", "words")
set6 = c("some", "many", "none", "few")

# difference between set5 and set6
```

```
setdiff(set5, set6)
```

```
## [1] "random" "words"
```

3.3.4 Set equality with setequal()

The function setequal() allows us to test the equality of two character vectors. If the vectors contain the same elements, setequal() returns TRUE (FALSE otherwise)

```
# three character vectors
set7 = c("some", "random", "strings")
set8 = c("some", "many", "none", "few")
set9 = c("strings", "random", "some")

# set7 == set8?
setequal(set7, set8)
```

```
## [1] FALSE
```

```
# set7 == set9?
setequal(set7, set9)
```

```
## [1] TRUE
```

3.3.5 Exact equality with identical()

Sometimes setequal() is not always what we want to use. It might be the case that we want to test whether two vectors are *exactly equal* (element by element). For instance, testing if set7 is exactly equal to set9. Although both vectors contain the same set of elements, they are not exactly the same vector. Such test can be performed with the function identical()

```
# set7 identical to set7?
identical(set7, set7)
```

```
## [1] TRUE
```

```
# set7 identical to set9?
identical(set7, set9)
```

```
## [1] FALSE
```

If we consult the help documentation of `identical()`, we can see that this function is the "safe and reliable way to test two objects for being exactly equal".

3.3.6 Element contained with `is.element()`

If we wish to test if an element is contained in a given set of character strings we can do so with `is.element()`:

```
# three vectors
set10 = c("some", "stuff", "to", "play", "with")
elem1 = "play"
elem2 = "crazy"

# elem1 in set10?
is.element(elem1, set10)

## [1] TRUE

# elem2 in set10?
is.element(elem2, set10)

## [1] FALSE
```

Alternatively, we can use the binary operator `%in%` to test if an element is contained in a given set. The function `%in%` returns `TRUE` if the first operand is contained in the second, and it returns `FALSE` otherwise:

```
# elem1 in set10?
elem1 %in% set10

## [1] TRUE

# elem2 in set10?
elem2 %in% set10

## [1] FALSE
```

3.3.7 Sorting with `sort()`

The function `sort()` allows us to sort the elements of a vector, either in increasing order (by default) or in decreasing order using the argument `decreasing`:

```
set11 = c("today", "produced", "example", "beautiful", "a", "nicely")

# sort (decreasing order)
sort(set11)

## [1] "a"         "beautiful" "example"   "nicely"
## [5] "produced"  "today"

# sort (increasing order)
sort(set11, decreasing=TRUE)

## [1] "today"     "produced"  "nicely"    "example"
## [5] "beautiful" "a"
```

If we have alpha-numeric strings, `sort()` will put the numbers first when sorting in increasing order:

```
set12 = c("today", "produced", "example", "beautiful", "1", "nicely")

# sort (decreasing order)
sort(set12)

## [1] "1"         "beautiful" "example"   "nicely"
## [5] "produced"  "today"

# sort (increasing order)
sort(set12, decreasing = TRUE)

## [1] "today"     "produced"  "nicely"    "example"
## [5] "beautiful" "1"
```

3.3.8 Repetition with `rep()`

A very common operation with strings is replication, that is, given a string we want to replicate it several times. Although there is no single function in R for that purpose, we can combine `paste()` and `rep()` like so:

```
# repeat "x" 4 times
paste(rep("x", 4), collapse = '')

## [1] "xxxx"
```