Excellent! Not basic, but not hard either. Heavy reliance on Wickham's stringr package. Examples in back of book are wonderful. Not too much regex.

Handling and Processing Strings in R

Gaston Sanchez
www.gastonsanchez.com

About this ebook

Abstract

This ebook aims to help you get started with manipulating strings in R. Although there are a few issues with R about string processing, some of us argue that R can be very well used for computing with character strings and text. R may not be as rich and diverse as other scripting languages when it comes to string manipulation, but it can take you very far if you know how. Hopefully this text will provide you enough material to do more advanced string and text processing operations.

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.

License

Citation

You can cite this work as:

Sanchez, G. (2013) **Handling and Processing Strings in R**
Trowchez Editions. Berkeley, 2013.

http://www.gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf

Revision

Version 1.3 (March, 2014)

Contents

Preface

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 string manipulation. 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.

Documentation on how to manipulate strings and text data in R is very scarce. This is mostly because R is not perceived as a scripting language (like Python or Java, among others). However, I seriously think that we need to have more available resources about this indispensable topic. This work is my two cents in this largely forgotten area.

Gaston Sanchez
Nantes, France
September 2013

Chapter 1

Introduction

Handling and processing text 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.

I'm sure there will be people telling you that *the fact that you can do something in R does not mean that you should do it.* And I agree (to some extent). Yes, you probably shouldn't use R for those tasks that can be better performed with languages like Python, Perl or Ruby. However, there will be many occasions in which it's better to stay inside the R environment (even if it is just for convenient reasons or for procrastination within R). I may definitely use Python if I have to, but whenever possible, I will try to stay with R (my personal taste).

1.1 Some Resources

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 for manipulating strings and text data.

Perhaps 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.
- Don't know this one
- 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

1.2 Character Strings and Data Analysis

This book aims to help you get started with handling strings in R. It provides an overview of several resources that you can use for string manipulation. It covers useful functions, general topics, common operations, and other tricks.

This book is NOT about textual data analysis, linguistic analysis, text mining, or natural language processing. For those purposes, 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

However, even if you don't plan to do text analysis, text mining, or natural language processing, I bet you have some dataset that contains data as characters: names of rows, names of columns, dates, monetary quantities, longitude and latitude, etc. 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

Hopefully 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.

1.3 A Toy Example

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. We'll use the data frame `USArrests` that already comes with R for this crash informal introduction. 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.

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" "Flor" "Geor"
## [11] "Hawa" "Idah" "Illi" "Indi" "Iowa" "Kans" "Kent" "Loui" "Main" "Mary"
## [21] "Mass" "Mich" "Minn" "Miss" "Miss" "Mont" "Nebr" "Neva" "New " "New "
## [31] "New " "New " "Nort" "Nort" "Ohio" "Okla" "Oreg" "Penn" "Rhod" "Sout"
## [41] "Sout" "Tenn" "Texa" "Utah" "Verm" "Virg" "Wash" "West" "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" "Flrd" "Gerg"
## [11] "Hawa" "Idah" "Illn" "Indn" "Iowa" "Knss" "Kntc" "Losn" "Main" "Mryl"
## [21] "Mssc" "Mchg" "Mnns" "Msss" "Mssr" "Mntn" "Nbrs" "Nevd" "NwHm" "NwJr"
## [31] "NwMx" "NwYr" "NrtC" "NrtD" "Ohio" "Oklh" "Orgn" "Pnns" "RhdI" "SthC"
## [41] "SthD" "Tnns" "Texs" "Utah" "Vrmn" "Vrgn" "Wshn" "WstV" "Wscn" "Wymn"
```

If we decide to try an abbreviation with six letters we just simply change the argument `min.ength = 5`

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

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

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)

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

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

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"     "Nebraska"
## [5] "New York"     "North Dakota" "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"          "New Hampshire"
## [5] "New Jersey"    "New Mexico"    "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"          "New Hampshire"
##  [5] "New Jersey"    "New Mexico"    "New York"      "Washington"
##  [9] "West Virginia" "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"          "new hampshire"
##  [5] "new jersey"    "new mexico"    "new york"      "washington"
##  [9] "west virginia" "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"          "NEW HAMPSHIRE"
##  [5] "NEW JERSEY"    "NEW MEXICO"    "NEW YORK"      "WASHINGTON"
##  [9] "WEST VIRGINIA" "WISCONSIN"     "WYOMING"
```

A third solution, and perhaps the simplest one, is to specify the argument `ignore.case=TRUE` inside `grep()`:

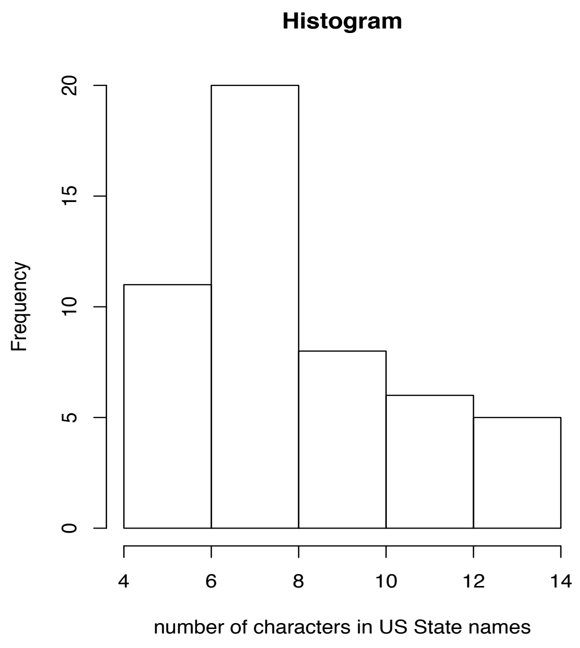
```
# get states names with 'w'
grep(pattern = "w", x = states, value = TRUE, ignore.case = TRUE)
```

```
##  [1] "Delaware"      "Hawaii"        "Iowa"          "New Hampshire"
##  [5] "New Jersey"    "New Mexico"    "New York"      "Washington"
##  [9] "West Virginia" "Wisconsin"     "Wyoming"
```

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:

```
# histogram
hist(nchar(states), 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 1 0 0 0 2 2 0
## [36] 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 1 0 0 0 2 2 0
## [36] 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:

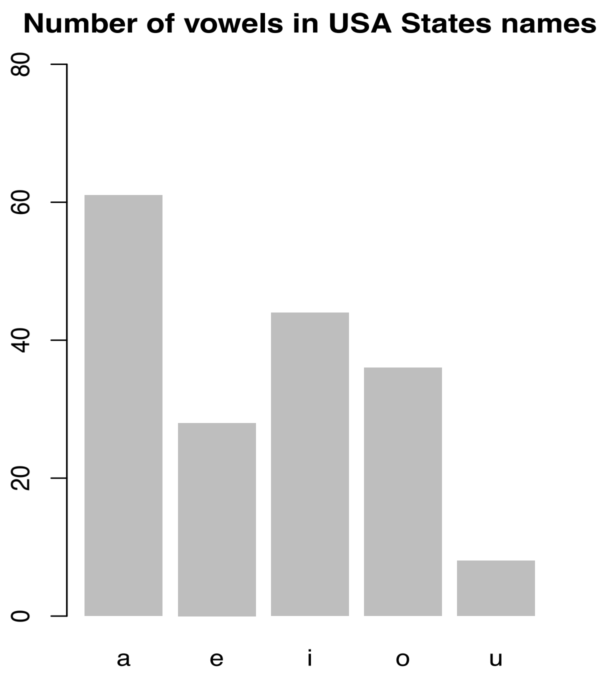
```
# total number of a's
str_count(tolower(states), "a")

##  [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 1 0 0 0 2 2 0
## [36] 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:

```
# vector of vowels
vowels = c("a", "e", "i", "o", "u")

# vector for storing results
num_vowels = vector(mode = "integer", length = 5)

# calculate number of vowels in each name
for (j in seq_along(vowels)) {
    num_aux = str_count(tolower(states), vowels[j])
    num_vowels[j] = sum(num_aux)
}

# add vowel names
names(num_vowels) = vowels

# total number of vowels
num_vowels

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


# sort them in decreasing order
sort(num_vowels, decreasing = TRUE)

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

And finally, we can visualize the distribution with a barplot:

```
# barplot
barplot(num_vowels, main = "Number of vowels in USA States names",
        border = NA, ylim = c(0, 80))
```



1.4 Overview

The previous examples with the names of USA States are just an appetizer of the menu contained in this book. The rest of this book is divided in six more chapters:

Chapter 2: Character Strings in R

Chapter 3: String Manipulations

Chapter 4: String manipulations with `stringr`

Chapter 5: Regular Expressions (part I)

Chapter 6: Regular Expressions (part II)

Chapter 7: Practical Applications

Chapter 2 is dedicated to show you the basics in R for working with character strings. We see how to get text data in R and how R objects behave when dealing with characters.

Chapter 3 describes a wide range of functions that can be used to manipulate strings without requiring regular expressions. The idea is to cover those functions within the base distribution of R.

Chapter 4 is intended to show you the string manipulation functions of the R package `stringr`. Similar to chapter 3, the content of chapter 4 is limited to those functions for working without regular expressions.

Chapter 5 is the first part of the discussion on regular expressions in R. Basically, the purpose of this chapter is to talk about the way R works with regular expressions (as compared to other scripting languages).

Chapter 6 continues the discussion around regular expressions. It covers the application of the regex functions in the base package as well as in the `stringr` package.

Chapter 7 is the last chapter in which we discuss some examples with practical applications. We get to apply part of the material covered in the book with simple exercises that are intended to motivate you in the potential of R for processing character strings.

Chapter 2

Character Strings in R

The main type of data R gives us to process text strings is *character*. Formally, the class of object that holds character strings in R is `"character"`. We express character strings using single or double quotes:

```
'a character string using single quotes'

"a character string using double quotes"
```

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'
```

We cannot 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'
```

2.1 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"`.

2.1.1 Empty string

Let's start with 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"
```

2.1.2 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.

2.1.3 `character()`

As we already mentioned, `character()` is a function to create character vectors. We just have to specify the length of the vector, and `character()` will produce a character vector with as many empty strings, for instance:

```
# character vector with 5 empty strings
char_vector = character(5)

# display
char_vector

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

Once an empty character object has been created, new components may be added to it simply by 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
```

We 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 we went from a one-element vector to a four-element vector without specifying the second and third elements. R fills this gap with missing values `NA`.

2.1.4 `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 and testing for objects of type `"character"`. To test if an object is of type `"character"` you use the function `is.character()`:

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

# 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"
```

For better or worse, R allows you to convert non-character objects into character strings with the function `as.character()`:

```
# converting 'b' as character
b = as.character(b)
b

## [1] "17"
```

2.2 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).

2.2.1 Behavior of R objects with character strings

Vectors The main, and most basic, type of objects in R are vectors. Vectors must have their values *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"
```

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] -0.6958 -2.2614 -1.4495  1.5161  1.3733
```

2.3 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)

2.3.1 Reading tables

If the data we want to import is in some tabular format 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

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()`:

```
# abc radio stations data URL
abc = "http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv"

# read data from URL
radio = read.table(abc, 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  "612 ABC 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  " 612brisbane" ...
##  $ Facebook        : chr  " http://www.facebook.com/612ABCBrisbane" ...
```

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.

2.3.2 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 `readLines()`. This function is the one we should use if we don't want R to assume that the data is 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, containing the contents of each line.

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

String Manipulations

In the previous chapter we talked about how R treats objects with characters, and how we can import text data. Now we will cover some of the basic (and not so basic) functions for manipulating character strings.

3.1 The versatile `paste()` function

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"
```

3.2 Printing characters

R provides a series of functions for printing strings. Some of the printing functions are useful when creating `print` methods for programmed objects' classes. Other functions are useful for printing outputs either in the R console or in a given file. In this section we will describe the following print-related functions:

Printing functions

Function	Description
`print()`	generic printing
`noquote()`	print with no quotes
`cat()`	concatenation
`format()`	special formats
`toString()`	convert to string
`sprintf()`	printing

3.2.1 Printing values with `print()`

The *workhorse* printing function in R is `print()`. As its names indicates, this function prints its argument on the R console:

```
# text string
my_string = "programming with data is fun"

# print string
print(my_string)

## [1] "programming with data is fun"
```

To be more precise, `print()` is a generic function, which means that you should use this function when creating printing methods for programmed classes.

As you can see from the previous example, `print()` displays text in quoted form by default. If we want to print character strings with no quotes we can set the argument `quote = FALSE`

```
# print without quotes
print(my_string, quote = FALSE)

## [1] programming with data is fun
```

The output produced by `print()` can be customized with various optional arguments. However, the way in which `print()` displays objects is rather limited. To get more printing variety there is a number of more flexible functions that we can use.

3.2.2 Unquoted characters with noquote()

We know that we can print text without quotes using `print()` with its argument `quote = FALSE`. An alternative option for achieving a similar output is by using `noquote()`. As its names implies, this function prints character strings with no quotes:

```
# noquote
noquote(my_string)

## [1] programming with data is fun
```

To be more precise `noquote()` creates a character object of class `"noquote"` which always gets displayed without quotes:

```
# class noquote
no_quotes = noquote(c("some", "quoted", "text", "!%^(&="))

# display
no_quotes

## [1] some    quoted text   !%^(&=

# check class
class(no_quotes)

## [1] "noquote"

# test character
is.character(no_quotes)

## [1] TRUE

# no quotes even when subscripting
no_quotes[2:3]

## [1] quoted text
```

3.2.3 Concatenate and print with cat()

Another very useful function is `cat()` which allows us to concatenate objects and print them either on screen or to a file. Its usage has the following structure:

```
cat(..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
```

The argument ... implies that `cat()` accepts several types of data. However, when we pass numeric and/or complex elements they are automatically converted to character strings by cat(). By default, the strings are concatenated with a space character as separator. This can be modified with the sep argument.

If we use cat() with only one single string, you get a similar (although not identical) result as noquote():

```
# simply print with 'cat()'
cat(my_string)

## programming with data is fun
```

As you can see, cat() prints its arguments without quotes. In essence, cat() simply displays its content (on screen or in a file). Compared to noquote(), cat() does not print the numeric line indicator ([1] in this case).

The usefulness of cat() is when we have two or more strings that we want to concatenate:

```
# concatenate and print
cat(my_string, "with R")

## programming with data is fun with R
```

You can use the argument sep to indicate a chacracter vector that will be included to separate the concatenated elements:

```
# especifying 'sep'
cat(my_string, "with R", sep = " =) ")

## programming with data is fun =) with R

# another example
cat(1:10, sep = "-")

## 1-2-3-4-5-6-7-8-9-10
```

When we pass vectors to cat(), each of the elements are treated as though they were separate arguments:

```
# first four months
cat(month.name[1:4], sep = " ")

## January February March April
```

The argument fill allows us to break long strings; this is achieved when we specify the string width with an integer number:

```
# fill = 30
cat("Loooooooooong strings", "can be displayed", "in a nice format",
    "by using the 'fill' argument", fill = 30)
```

```
## Looooooooooong strings
## can be displayed
## in a nice format
## by using the 'fill' argument
```

Last but not least, we can specify a file output in `cat()`. For instance, let's suppose that we want to save the output in the file `output.txt` located in our working directory:

```
# cat with output in a given file
cat(my_string, "with R", file = "output.txt")
```

3.2.4 Encoding strings with `format()`

The function `format()` allows us to format an R object for pretty printing. Essentially, `format()` treats the elements of a vector as character strings using a common format. This is especially useful when printing numbers and quantities under different formats.

```
# default usage
format(13.7)

## [1] "13.7"

# another example
format(13.12345678)

## [1] "13.12"
```

Some useful arguments:

- `width` the (minimum) width of strings produced
- `trim` if set to `TRUE` there is no padding with spaces
- `justify` controls how padding takes place for strings. Takes the values `"left"`, `"right"`, `"centre"`, `"none"`

For controling the printing of numbers, use these arguments:

- `digits` The number of digits to the right of the decimal place.
- `scientific` use `TRUE` for scientific notation, `FALSE` for standard notation

Keep in mind that `justify` does not apply to numeric values.

```
# use of 'nsmall'
format(13.7, nsmall = 3)

## [1] "13.700"
```

```
# use of 'digits'
format(c(6, 13.1), digits = 2)

## [1] " 6" "13"

# use of 'digits' and 'nsmall'
format(c(6, 13.1), digits = 2, nsmall = 1)

## [1] " 6.0" "13.1"
```

By default, `format()` pads the strings with spaces so that they are all the same length.

```
# justify options
format(c("A", "BB", "CCC"), width = 5, justify = "centre")

## [1] "  A  " " BB  " " CCC "

format(c("A", "BB", "CCC"), width = 5, justify = "left")

## [1] "A    " "BB   " "CCC  "

format(c("A", "BB", "CCC"), width = 5, justify = "right")

## [1] "    A" "   BB" "  CCC"

format(c("A", "BB", "CCC"), width = 5, justify = "none")

## [1] "A"   "BB"  "CCC"


# digits
format(1/1:5, digits = 2)

## [1] "1.00" "0.50" "0.33" "0.25" "0.20"


# use of 'digits', widths and justify
format(format(1/1:5, digits = 2), width = 6, justify = "c")

## [1] " 1.00 " " 0.50 " " 0.33 " " 0.25 " " 0.20 "
```

For printing large quantities with a sequenced format we can use the arguments `big.mark` or `big.interval`. For instance, here is how we can print a number with sequences separated by a comma ","

```
# big.mark
format(123456789, big.mark = ",")

## [1] "123,456,789"
```

3.2.5 C-style string formatting with `sprintf()`

The function `sprintf()` is a wrapper for the `C` function `sprintf()` that returns a formatted string combining text and variable values. The nice feature about `sprintf()` is that it provides us a very flexible way of formatting vector elements as character strings. Its usage has the following form:

```
sprintf(fmt, ...)
```

The argument `fmt` is a character vector of format strings. The allowed conversion specifications start the symbol `%` followed by numbers and letters. For demonstration purposes here are several ways in which the number `pi` can be formatted:

```
# '%f' indicates 'fixed point' decimal notation
sprintf("%f", pi)

## [1] "3.141593"

# decimal notation with 3 decimal digits
sprintf("%.3f", pi)

## [1] "3.142"

# 1 integer and 0 decimal digits
sprintf("%1.0f", pi)

## [1] "3"

# decimal notation with 3 decimal digits
sprintf("%5.1f", pi)

## [1] "  3.1"

sprintf("%05.1f", pi)

## [1] "003.1"

# print with sign (positive)
sprintf("%+f", pi)

## [1] "+3.141593"

# prefix a space
sprintf("% f", pi)

## [1] " 3.141593"

# left adjustment
sprintf("%-10f", pi)  # left justified
```

```
## [1] "3.141593  "

# exponential decimal notation 'e'
sprintf("%e", pi)

## [1] "3.141593e+00"

# exponential decimal notation 'E'
sprintf("%E", pi)

## [1] "3.141593E+00"

# number of significant digits (6 by default)
sprintf("%g", pi)

## [1] "3.14159"
```

3.2.6 Converting objects to strings with `toString()`

The function `toString()` allows us to convert an R object to a character string. This function can be used as a helper for `format()` to produce a single character string from several obejcts inside a vector. The result will be a character vector of length 1 with elements separated by commas:

```
# default usage
toString(17.04)

## [1] "17.04"

# combining two objects
toString(c(17.04, 1978))

## [1] "17.04, 1978"

# combining several objects
toString(c("Bonjour", 123, TRUE, NA, log(exp(1))))

## [1] "Bonjour, 123, TRUE, NA, 1"
```

One of the nice features about `toString()` is that you can specify its argument `width` to fix a maximum field width.

```
# use of 'width'
toString(c("one", "two", "3333333333"), width = 8)

## [1] "one,...."

# use of 'width'
```

```
toString(c("one", "two", "3333333333"), width = 12)

## [1] "one, two...."
```

3.2.7 Comparing printing methods

Even though R has just a small collection of functions for printing and formatting strings, we can use them to get a wide variety of outputs. The choice of function (and its arguments) will depend on *what* we want to print, *how* we want to print it, and *where* we want to print it. Sometimes the answer of which function to use is straightforward. Sometimes however, we would need to experiment and compare different ways until we find the most adequate method. To finish this section let's consider a simple example with a numeric vector with 5 elements:

```
# printing method
print(1:5)

## [1] 1 2 3 4 5

# convert to character
as.character(1:5)

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

# concatenation
cat(1:5, sep = "-")

## 1-2-3-4-5

# default pasting
paste(1:5)

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

# paste with collapsing
paste(1:5, collapse = "")

## [1] "12345"

# convert to a single string
toString(1:5)

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

# unquoted output
noquote(as.character(1:5))
```

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

3.3 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

3.3.1 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.

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

## [1] 3


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

## [1] 1
```

3.3.2 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:

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

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

3.3.3 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:

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

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

3.3.4 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 (characters in upper case):

```r
# 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"
```

3.3.5 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: '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"
```

3.3.6 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"  "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"
```

3.3.7 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"
```

3.3.8 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"     "lezz"
```

3.4 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

3.4.1 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.

3.4.2 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"
```

3.4.3 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"
```

3.4.4 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
```

3.4.5 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".

3.4.6 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
```

3.4.7 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"    "produced"  "today"


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

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


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

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

3.4.8 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"
```

Chapter 4

String manipulations with `stringr`

As we saw in the previous chapters, R provides a useful range of functions for basic string processing and manipulations of `"character"` data. Most of the times these functions are enough and they will allow us to get our job done. However, they have some drawbacks. For instance, consider the following example:

```
# some text vector
text = c("one", "two", "three", NA, "five")

# how many characters in each string?
nchar(text)

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

As you can see, `nchar()` gives `NA` a value of 2, as if it were a string formed by two characters. Perhaps this may be acceptable in some cases, but taking into account all the operations in R, it would be better to leave `NA` as is, instead of treating it as a string of two characters.

Another awkward example can be found with `paste()`. The default separator is a blank space, which more often than not is what we want to use. But that's secondary. The really annoying thing is when we want to paste things that include zero length arguments. How does `paste()` behave in those cases? See below:

```
# this works fine
paste("University", "of", "California", "Berkeley")

## [1] "University of California Berkeley"


# this works fine too
paste("University", "of", "California", "Berkeley")
```

```
## [1] "University of California Berkeley"


# this is weird
paste("University", "of", "California", "Berkeley", NULL)

## [1] "University of California Berkeley "


# this is ugly
paste("University", "of", "California", "Berkeley", NULL, character(0),
      "Go Bears!")

## [1] "University of California Berkeley   Go Bears!"
```

Notice the output from the last example (the *ugly* one). The objects `NULL` and `character(0)` have zero length, yet when included inside `paste()` they are treated as an empty string `""`. Wouldn't be good if `paste()` removed zero length arguments? Sadly, there's nothing we can do to change `nchar()` and `paste()`. But fear not. There is a very nice package that solves these problems and provides several functions for carrying out consistent string processing.

4.1 Package `stringr`

Thanks to Hadley Wickham, we have the package `stringr` that adds more functionality to the base functions for handling strings in R. According to the description of the package (see http://cran.r-project.org/web/packages/stringr/index.html) `stringr`

> "is a set of simple wrappers that make R's string functions more consistent, simpler and easier to use. It does this by ensuring that: function and argument names (and positions) are consistent, all functions deal with NA's and zero length character appropriately, and the output data structures from each function matches the input data structures of other functions."

To install `stringr` use the function `install.packages()`. Once installed, load it to your current session with `library()`:

```
# installing 'stringr'
install.packages("stringr")

# load 'stringr'
library(stringr)
```

4.2 Basic String Operations

`stringr` provides functions for both 1) basic manipulations and 2) for regular expression operations. In this chapter we cover those functions that have to do with basic manipulations. In turn, regular expression functions with `stringr` are discussed in chapter 6.

The following table contains the `stringr` functions for basic string operations:

Function	Description	Similar to
`str_c()`	string concatenation	`paste()`
`str_length()`	number of characters	`nchar()`
`str_sub()`	extracts substrings	`substring()`
`str_dup()`	duplicates characters	*none*
`str_trim()`	removes leading and trailing whitespace	*none*
`str_pad()`	pads a string	*none*
`str_wrap()`	wraps a string paragraph	`strwrap()`
`str_trim()`	trims a string	*none*

As you can see, all functions in `stringr` start with `"str_"` followed by a term associated to the task they perform. For example, `str_length()` gives us the number (i.e. length) of characters in a string. In addition, some functions are designed to provide a better alternative to already existing functions. This is the case of `str_length()` which is intended to be a substitute of `nchar()`. Other functions, however, don't have a corresponding alternative such as `str_dup()` which allows us to duplicate characters.

4.2.1 Concatenating with `str_c()`

Let's begin with `str_c()`. This function is equivalent to `paste()` but instead of using the white space as the default separator, `str_c()` uses the empty string `""`.

```
# default usage
str_c("May", "The", "Force", "Be", "With", "You")

## [1] "MayTheForceBeWithYou"


# removing zero length objects
str_c("May", "The", "Force", NULL, "Be", "With", "You", character(0))

## [1] "MayTheForceBeWithYou"
```

Notice another major difference between `str_c()` and `paste()`: zero length arguments like `NULL` and `character(0)` are silently removed by `str_c()`.

If we want to change the default separator, we can do that as usual by specifying the argument `sep`:

```
# changing separator
str_c("May", "The", "Force", "Be", "With", "You", sep = "_")

## [1] "May_The_Force_Be_With_You"


# synonym function 'str_join'
str_join("May", "The", "Force", "Be", "With", "You", sep = "-")

## [1] "May-The-Force-Be-With-You"
```

As you can see from the previous examples, a synonym for `str_c()` is `str_join()`.

4.2.2 Number of characters with `str_length()`

As we've mentioned before, the function `str_length()` is equivalent to `nchar()`. Both functions return the number of characters in a string, that is, the *length* of a string (do not confuse it with the `length()` of a vector). Compared to `nchar()`, `str_length()` has a more consistent behavior when dealing with `NA` values. Instead of giving `NA` a length of 2, `str_length()` preserves missing values just as `NA`s.

```
# some text (NA included)
some_text = c("one", "two", "three", NA, "five")

# compare 'str_length' with 'nchar'
nchar(some_text)

## [1] 3 3 5 2 4

str_length(some_text)

## [1]  3  3  5 NA  4
```

In addition, `str_length()` has the nice feature that it converts factors to characters, something that `nchar()` is not able to handle:

```
# some factor
some_factor = factor(c(1, 1, 1, 2, 2, 2), labels = c("good", "bad"))
some_factor

## [1] good good good bad  bad  bad
## Levels: good bad

# try 'nchar' on a factor
```

```
nchar(some_factor)

## Error:  'nchar()' requires a character vector

# now compare it with 'str_length'
str_length(some_factor)

## [1] 4 4 4 3 3 3
```

4.2.3 Substring with `str_sub()`

To extract substrings from a character vector `stringr` provides `str_sub()` which is equivalent to `substring()`. The function `str_sub()` has the following usage form:

```
 str_sub(string, start = 1L, end = -1L)
```

The three arguments in the function are: a `string` vector, a `start` value indicating the position of the first character in substring, and an `end` value indicating the position of the last character. Here's a simple example with a single string in which characters from 1 to 5 are extracted:

```
# some text
lorem = "Lorem Ipsum"

# apply 'str_sub'
str_sub(lorem, start = 1, end = 5)

## [1] "Lorem"

# equivalent to 'substring'
substring(lorem, first = 1, last = 5)

## [1] "Lorem"


# another example
str_sub("adios", 1:3)

## [1] "adios" "dios"  "ios"
```

An interesting feature of `str_sub()` is its ability to work with negative indices in the `start` and `end` positions. When we use a negative position, `str_sub()` counts backwards from last character:

```
# some strings
resto = c("brasserie", "bistrot", "creperie", "bouchon")
```

```
# 'str_sub' with negative positions
str_sub(resto, start = -4, end = -1)

## [1] "erie" "trot" "erie" "chon"

# compared to substring (useless)
substring(resto, first = -4, last = -1)

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

Similar to `substring()`, we can also give `str_sub()` a set of positions which will be recycled over the string. But even better, we can give `str_sub()` a negative sequence, something that `substring()` ignores:

```
# extracting sequentially
str_sub(lorem, seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "orem Ipsum"  "rem Ipsum"   "em Ipsum"    "m Ipsum"
##  [6] " Ipsum"      "Ipsum"       "psum"        "sum"         "um"
## [11] "m"

substring(lorem, seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "orem Ipsum"  "rem Ipsum"   "em Ipsum"    "m Ipsum"
##  [6] " Ipsum"      "Ipsum"       "psum"        "sum"         "um"
## [11] "m"

# reverse substrings with negative positions
str_sub(lorem, -seq_len(nchar(lorem)))

##  [1] "m"           "um"          "sum"         "psum"        "Ipsum"
##  [6] " Ipsum"      "m Ipsum"     "em Ipsum"    "rem Ipsum"   "orem Ipsum"
## [11] "Lorem Ipsum"

substring(lorem, -seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
##  [6] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
## [11] "Lorem Ipsum"
```

We can use `str_sub()` not only for extracting subtrings but also for replacing substrings:

```
# replacing 'Lorem' with 'Nullam'
lorem = "Lorem Ipsum"
str_sub(lorem, 1, 5) <- "Nullam"
lorem

## [1] "Nullam Ipsum"
```

```
# replacing with negative positions
lorem = "Lorem Ipsum"
str_sub(lorem, -1) <- "Nullam"
lorem

## [1] "Lorem IpsuNullam"


# multiple replacements
lorem = "Lorem Ipsum"
str_sub(lorem, c(1, 7), c(5, 8)) <- c("Nullam", "Enim")
lorem

## [1] "Nullam Ipsum"  "Lorem Enimsum"
```

4.2.4 Duplication with `str_dup()`

A common operation when handling characters is *duplication*. The problem is that R doesn't have a specific function for that purpose. But `stringr` does: `str_dup()` duplicates and concatenates strings within a character vector. Its usage requires two arguments:

```
 str_dup(string, times)
```

The first input is the `string` that we want to repeat. The second input, `times`, is the number of times to duplicate each string:

```
# default usage
str_dup("hola", 3)

## [1] "holaholahola"


# use with differetn 'times'
str_dup("adios", 1:3)

## [1] "adios"           "adiosadios"      "adiosadiosadios"


# use with a string vector
words = c("lorem", "ipsum", "dolor", "sit", "amet")
str_dup(words, 2)

## [1] "loremlorem" "ipsumipsum" "dolordolor" "sitsit"     "ametamet"

str_dup(words, 1:5)
```

```
## [1] "lorem"                  "ipsumipsum"             "dolordolordolor"
## [4] "sitsitsitsit"           "ametametametametamet"
```

4.2.5 Padding with `str_pad()`

Another handy function that we can find in `stringr` is `str_pad()` for *padding* a string. Its default usage has the following form:

```
str_pad(string, width, side = "left", pad = " ")
```

The idea of `str_pad()` is to take a string and pad it with leading or trailing characters to a specified total `width`. The default padding character is a space (`pad = " "`), and consequently the returned string will appear to be either left-aligned (`side = "left"`), right-aligned (`side = "right"`), or both (`side = "both"`)

Let's see some examples:

```
# default usage
str_pad("hola", width = 7)

## [1] "   hola"


# pad both sides
str_pad("adios", width = 7, side = "both")

## [1] " adios "


# left padding with '#'
str_pad("hashtag", width = 8, pad = "#")

## [1] "#hashtag"


# pad both sides with '-'
str_pad("hashtag", width = 9, side = "both", pad = "-")

## [1] "-hashtag-"
```

4.2.6 Wrapping with `str_wrap()`

The function `str_wrap()` is equivalent to `strwrap()` which can be used to *wrap* a string to format paragraphs. The idea of wrapping a (long) string is to first split it into paragraphs according to the given `width`, and then add the specified indentation in each line (first line with `indent`, following lines with `exdent`). Its default usage has the following form:

```
str_wrap(string, width = 80, indent = 0, exdent = 0)
```

For instance, consider the following quote (from Douglas Adams) converted into a paragraph:

```
# quote (by Douglas Adams)
some_quote = c(
  "I may not have gone",
  "where I intended to go,",
  "but I think I have ended up",
  "where I needed to be")

# some_quote in a single paragraph
some_quote = paste(some_quote, collapse = " ")
```

Now, say we want to display the text of `some_quote` within some pre-specified column width (e.g. width of 30). We can achieve this by applying `str_wrap()` and setting the argument `width = 30`

```
# display paragraph with width=30
cat(str_wrap(some_quote, width = 30))

## I may not have gone where I
## intended to go, but I think I
## have ended up where I needed
## to be
```

Besides displaying a (long) paragraph into several lines, we may also wish to add some indentation. Here's how we can indent the first line, as well as the following lines:

```
# display paragraph with first line indentation of 2
cat(str_wrap(some_quote, width = 30, indent = 2), "\n")

##   I may not have gone where I
## intended to go, but I think I
## have ended up where I needed
## to be


# display paragraph with following lines indentation of 3
cat(str_wrap(some_quote, width = 30, exdent = 3), "\n")

## I may not have gone where I
##    intended to go, but I
##    think I have ended up
##    where I needed to be
```

4.2.7 Trimming with `str_trim()`

One of the typical tasks of string processing is that of parsing a text into individual words. Usually, we end up with words that have blank spaces, called *whitespaces*, on either end of the word. In this situation, we can use the `str_trim()` function to remove any number of whitespaces at the ends of a string. Its usage requires only two arguments:

```
str_trim(string, side = "both")
```

The first input is the `string` to be strimmed, and the second input indicates the `side` on which the whitespace will be removed.

Consider the following vector of strings, some of which have whitespaces either on the left, on the right, or on both sides. Here's what `str_trim()` would do to them under different settings of `side`

```
# text with whitespaces
bad_text = c("This", " example ", "has several   ", "   whitespaces ")

# remove whitespaces on the left side
str_trim(bad_text, side = "left")

## [1] "This"           "example "       "has several   " "whitespaces "

# remove whitespaces on the right side
str_trim(bad_text, side = "right")

## [1] "This"           " example"       "has several"    "   whitespaces"

# remove whitespaces on both sides
str_trim(bad_text, side = "both")

## [1] "This"        "example"     "has several" "whitespaces"
```

4.2.8 Word extraction with `word()`

We end this chapter describing the `word()` function that is designed to extract words from a sentence:

```
word(string, start = 1L, end = start, sep = fixed(" "))
```

The way in which we use `word()` is by passing it a `string`, together with a `start` position of the first word to extract, and an `end` position of the last word to extract. By default, the separator `sep` used between words is a single space.

Let's see some examples:

```
# some sentence
change = c("Be the change", "you want to be")

# extract first word
word(change, 1)

## [1] "Be"  "you"


# extract second word
word(change, 2)

## [1] "the"  "want"


# extract last word
word(change, -1)

## [1] "change" "be"


# extract all but the first words
word(change, 2, -1)

## [1] "the change" "want to be"
```

`stringr` has more functions but we'll discuss them in chapter 6 since they have to do with regular expressions.

Chapter 5

Regular Expressions (part I)

So far we have seen some basic and intermediate functions for handling and working with text in R. These are very useful functions and they allows us to do many interesting things. However, if we truly want to unleash the power of strings manipulation, we need to take things to the next level and talk about *regular expressions*.

A **regular expression** (a.k.a. **regex**) is a special text string for describing a certain amount of text. This "certain amount of text" receives the formal name of **pattern**. Hence we say that a regular expression is a *pattern that describes a set of strings*.

Tools for working with regular expressions can be found in virtually all scripting languages (e.g. Perl, Python, Java, Ruby, etc). R has some functions for working with regular expressions although it does not provide the wide range of capabilities that other scripting languages do. Nevertheless, they can take us very far with some workarounds (and a bit of patience).

I am assuming that you already have been introduced to regex, so we won't cover everything there is to know about regular expressions. Instead, we will focus on how R works with regular expressions, as well as the R syntax that you will have to use for regex operations. To know more about regular expressions in general, you can find some useful information in the following resources:

- Regex wikipedia
  `http://en.wikipedia.org/wiki/Regular_expression`
  For those readers who have no experience with regular expressions, a good place to start is by checking the wikipedia entrance.

- Regular-Expressions.info website (by Jan Goyvaerts)
  `http://www.regular-expressions.info`
  An excelent website full of information about regular expressions. It contains many different topics, resources, lots of examples, and tutorials, covered at both beginner and

advanced levels.

- Mastering Regular Expressions (by Jeffrey Friedl)
  http://regex.info/book.html
  I wasn't sure whether to include this reference but I think it deserves to be considered as well. This is perhaps the authoritative book on regular expressions. The only issue is that it is a book better adressed for readers already experienced with regex.

5.1 Regex Basics

The main purpose of working with regular expressions is to describe patterns that are used to match against text strings. Simply put, working with regular expressions is nothing more than **pattern matching**. The result of a match is either successful or not.

The simplest version of pattern matching is to search for one occurrence (or all occurrences) of some specific characters in a string. For example, we might want to search for the word `"programming"` in a large text document, or we might want to search for all occurrences of the string `"apply"` in a series of files containing R scripts.

Typically, regular expression patterns consist of a combination of alphanumeric characters as well as special characters. A regex pattern can be as simple as a single character, or it can be formed by several characters with a more complex structure. In all cases we construct regular expressions much in the same form in which we construct arithmetic expressions, by using various operators to combine smaller expressions.

Among the various types of operators, their main use reduces to four basic operations for creating regular expressions:

- Concatenation
- Logical OR
- Replication
- Grouping

The operations listed above can be considered to be the building blocks of regular expressions operators. By combining them in several ways, we can represent very complicated and sophisticated patterns. Here is a short description of each of them:

Concatenation The basic type of regular expression is formed by concatenating a set of characters together, one after the other, like `"abcd"`. This regex pattern matches only the single string "abcd".

Logical OR The logical operator *OR*, denoted by the vertical bar `|`, allows us to choose from one of several possibilities. For example, the regular expression `"ab|cd"` matches exactly two strings “ab” and “cd”. With the logical operator we can specify many strings with a single regular expression. For instance, if we are trying to find among a bunch of documents, those texts related to hydrology, we might look for words like “water”, “ocean”, “sea”, “river” and “lake”. All those terms can be put together in the form of a regular expression such as `"water|ocean|sea|river|lake"`.

Repetition Another basic operation is that of repetition, which enables us to define a pattern that matches under multiple possibilities. More specifically, this operation is carried out using a series of regex operators, known as *quantifier*, that repeat the preceding regular expression a specified number of times.

Grouping The grouping operator, denoted with a expression inside parentheses `( )`, enables us to specify any number of other operators and characters as a unit. In other words, a grouping sequence is a parenthesized expression that is treated as a unit. For example, if we want to specify the set of strings X, XYX, XYXYX, XYXYXYX, and so forth, we must write `"(XY)*X"` to indicate that the “XY” pattern must be replicated together.

5.2 Regular Expressions in R

There are two main aspects that we need to consider about regular expressions in R. One has to do with *the functions* designed for regex pattern matching. The other aspect has to do with *the way* regex patterns are expressed in R. In this section we are going to talk about the latter issue: the way R works with regular expressions. I find more convenient to first cover the specificities of R around regex operations, before discussing the functions and how to interact with regex patterns.

5.2.1 Regex syntax details in R

Most of the regex manipulations that we would use in other scripting languages work in the same way in R. But not everything. Some regex elements and patterns require a specific synthax in R which we need to describe.

To know more about the specifications and technicalities of regular expressions as used in R, you should see the help pages: `help(regex)` or `help(regexp)` (or alternatively `?regex`).

```r
# to know more about regular expressions in R
help(regex)
```

The help documentation about regular expressions contains a lot of technical information. Depending on your level of regex expertise, you may find the provided information very useful or simply criptic. But don't expect to find any examples. The help content is just for reference with no practical demos on R. With respect to this book, we will cover the following topics in the next subsections:

- metacharacters
- quanitifiers
- sequences
- character classes
- POSIX character classes

5.2.2 Metacharacters

The simplest form of regular expressions are those that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. For example, the pattern `"1"` matches the number 1. The pattern `"="` matches the equal symbol. The pattern `"blu"` matches the set of letters "blu". However, there are some special characters that have a reserved status and they are known as **metacharacters**. The metacharacters in *Extended Regular Expressions* (EREs) are:

```
. \ | ( ) [ { $ * + ?
```

For example, the pattern `"money$"` does not match "money$". Likewise, the pattern `"what?"` does not match "what?". Except for a few cases, metacharacters have a special meaning and purpose when working with regular expressions.

Normally —outside R—, when we want to represent any of the metacharacters with its literal meaning in a regex pattern, we need to escape them with a backslash \. In R, however, we need to escape them with a double backslash \\. The following table shows the general regex metacharacters and how to escape them in R:

Metacharacters and how to escape them in R

Metacharacter	Literal meaning	Escape in R
.	the period or dot	\\.
$	the dollar sign	\\$
*	the asterisk or star	*
+	the plus sign	\\+
?	the question mark	\\?
|	the vertical bar or pipe symbol	\\\|
\	the backslash	\\\\
^	the caret	\\^
[	the opening square bracket	\\[
]	the closing square bracket	\\]
{	the opening curly bracket	\\{
}	the closing curly bracket	\\}
(	the opening round bracket	\\(
)	the closing round bracket	\\)

For instance, consider the character string `"$money"`. Say we wanted to replace the dollar sign `$` with an empty string `""`. This can be done with the function `sub()`. The naive (but wrong) way is to simply try to match the dollar sign with the character `"$"`:

```
# string
money = "$money"

# the naive but wrong way
sub(pattern = "$", replacement = "", x = money)

## [1] "$money"
```

As you can see, nothing happened. In most scripting languages, when we want to represent any of the metacharacters with its literal meaning in a regex, we need to escape them with a backslash. The problem with R is that the traditional way doesn't work (you will get a nasty error):

```
# the usual (in other languages) yet wrong way in R
sub(pattern = "\$", replacement = "", x = money)
```

In R we need to scape metacharacters with a double backslash. This is how we would effectively replace the `$` sign:

```
# the right way in R
sub(pattern = "\\$", replacement = "", x = money)

## [1] "money"
```

Here are some silly examples that show how to escape metacharacters in R in order to be replaced with an empty "":

```
# dollar
sub("\\$", "", "$Peace-Love")

## [1] "Peace-Love"

# dot
sub("\\.", "", "Peace.Love")

## [1] "PeaceLove"

# plus
sub("\\+", "", "Peace+Love")

## [1] "PeaceLove"

# caret
sub("\\^", "", "Peace^Love")

## [1] "PeaceLove"

# vertical bar
sub("\\|", "", "Peace|Love")

## [1] "PeaceLove"

# opening round bracket
sub("\\(", "", "Peace(Love)")

## [1] "PeaceLove)"

# closing round bracket
sub("\\)", "", "Peace(Love)")

## [1] "Peace(Love"

# opening square bracket
sub("\\[", "", "Peace[Love]")

## [1] "PeaceLove]"

# closing square bracket
sub("\\]", "", "Peace[Love]")

## [1] "Peace[Love"

# opening curly bracket
sub("\\{", "", "Peace{Love}")
## [1] "PeaceLove}"

# closing curly bracket
sub("\\}", "", "Peace{Love}")

## [1] "Peace{Love"

# double backslash
sub("\\\\", "", "Peace\\Love")

## [1] "PeaceLove"
```

5.2.3 Sequences

Sequences define, no surprinsingly, sequences of characters which can match. We have short-hand versions (or anchors) for commonly used sequences in R:

Anchor Sequences in R

Anchor	Description
`\\d`	match a digit character
`\\D`	match a non-digit character
`\\s`	match a space character
`\\S`	match a non-space character
`\\w`	match a word character
`\\W`	match a non-word character
`\\b`	match a word boundary
`\\B`	match a non-(word boundary)
`\\h`	match a horizontal space
`\\H`	match a non-horizontal space
`\\v`	match a vertical space
`\\V`	match a non-vertical space

Let's see an application of the anchor sequences with some examples around substitution operations. Substitutions can be performed with the functions `sub()` and `gsub()`. Although we'll discuss the replacements functions later in this chapter, it is important to keep in mind their difference. `sub()` replaces the *first* match, while `gsub()` replaces *all* the matches.

Let's consider the string `"the dandelion war 2010"`, and let's check the different substition effects of replacing each anchor sequence with an underscore `"_"`.

Digits and non-digits

```r
# replace digit with '_'
sub("\\d", "_", "the dandelion war 2010")

## [1] "the dandelion war _010"

gsub("\\d", "_", "the dandelion war 2010")

## [1] "the dandelion war ____"


# replace non-digit with '_'
sub("\\D", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\\D", "_", "the dandelion war 2010")

## [1] "__________________2010"
```

Spaces and non-spaces

```r
# replace space with '_'
sub("\\s", "_", "the dandelion war 2010")

## [1] "the_dandelion war 2010"

gsub("\\s", "_", "the dandelion war 2010")

## [1] "the_dandelion_war_2010"


# replace non-space with '_'
sub("\\S", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\\S", "_", "the dandelion war 2010")

## [1] "___ _________ ___ ____"
```

Words and non-words

```r
# replace word with '_'
sub("\\b", "_", "the dandelion war 2010")

## [1] "_the dandelion war 2010"

gsub("\\b", "_", "the dandelion war 2010")

## [1] "_t_h_e_ _d_a_n_d_e_l_i_o_n_ _w_a_r_ _2_0_1_0_"
```

```
# replace non-word with '_'
sub("\\B", "_", "the dandelion war 2010")

## [1] "t_he dandelion war 2010"

gsub("\\B", "_", "the dandelion war 2010")

## [1] "t_he d_an_de_li_on w_ar 2_01_0"
```

Word boundaries and non-word-boundaries

```
# replace word boundary with '_'
sub("\\w", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\\w", "_", "the dandelion war 2010")

## [1] "___ _________ ___ ____"


# replace non-word-boundary with '_'
sub("\\W", "_", "the dandelion war 2010")

## [1] "the_dandelion war 2010"

gsub("\\W", "_", "the dandelion war 2010")

## [1] "the_dandelion_war_2010"
```

5.2.4 Character Classes

A *character class* or *character set* is a list of characters enclosed by square brackets `[ ]`. Character sets are used to match **only one** of several characters. For instance, the regex character class `[aA]` matches any lower case letter `a` or any upper case letter `A`. Likewise, the regular expression `[0123456789]` matches any single digit. It is important not to confuse a **regex character class** with the native R `"character"` class concept.

A particular case of character classes is when we include the caret ^ at the beginning of the list: this indicates that the regular expression matches any character NOT in the list. This is the reason why this configuration is also known as **class negation** or negated character class. For example, the regular expression `[^xyz]` matches anything except the characters `x`, `y` or `z`.

In addition, sets of characters such as digits, lower case ASCII letters, and upper case ASCII letters can be specified as a range of characters by giving the first and last characters, separated by a hyphen. For instance, the regular expression `[a-z]` matches any lower case ASCII letter. In turn the chracter class `[0-9]` matches any digit.

Some (Regex) Character Classes

Anchor	Description
`[aeiou]`	match any one lower case vowel
`[AEIOU]`	match any one upper case vowel
`[0123456789]`	match any digit
`[0-9]`	match any digit (same as previous class)
`[a-z]`	match any lower case ASCII letter
`[A-Z]`	match any upper case ASCII letter
`[a-zA-Z0-9]`	match any of the above classes
`[^aeiou]`	match anything other than a lowercase vowel
`[^0-9]`	match anything other than a digit

Let's see a basic example. Imagine that we have a character vector with several words and that we are interested in matching those words containing the vowels `"e"` or `"i"`. For this purpose, we can use the character class `"[ei]"`:

```
# some string
transport = c("car", "bike", "plane", "boat")

# look for 'e' or 'i'
grep(pattern = "[ei]", transport, value = TRUE)

## [1] "bike"  "plane"
```

Here's another example with digit character classes:

```
# some numeric strings
numerics = c("123", "17-April", "I-II-III", "R 3.0.1")

# match strings with 0 or 1
grep(pattern = "[01]", numerics, value = TRUE)

## [1] "123"      "17-April" "R 3.0.1"

# match any digit
grep(pattern = "[0-9]", numerics, value = TRUE)

## [1] "123"      "17-April" "R 3.0.1"

# negated digit
grep(pattern = "[^0-9]", numerics, value = TRUE)

## [1] "17-April" "I-II-III" "R 3.0.1"
```

5.2.5 POSIX Character Classes

Closely related to the regex character classes we have what is known as *POSIX character classes*. In R, POSIX character classes are represented with expressions inside double brackets `[[ ]]`. The following table shows the POSIX character classes as used in R:

POSIX Character Classes in R

Class	Description
`[[:lower:]]`	Lower-case letters
`[[:upper:]]`	Upper-case letters
`[[:alpha:]]`	Alphabetic characters (`[[:lower:]]` and `[[:upper:]]`)
`[[:digit:]]`	Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`[[:alnum:]]`	Alphanumeric characters (`[[:alpha:]]` and `[[:digit:]]`)
`[[:blank:]]`	Blank characters: space and tab
`[[:cntrl:]]`	Control characters
`[[:punct:]]`	Punctuation characters: ! " # % & ' ( ) * + , - . / : ;
`[[:space:]]`	Space characters: tab, newline, vertical tab, form feed, carriage return, and space
`[[:xdigit:]]`	Hexadecimal digits: 0-9 A B C D E F a b c d e f
`[[:print:]]`	Printable characters (`[[:alpha:]]`, `[[:punct:]]` and space)
`[[:graph:]]`	Graphical characters (`[[:alpha:]]` and `[[:punct:]]`)

For example, suppose we are dealing with the following string:

```
# la vie (string)
la_vie = "La vie en #FFC0CB (rose);\nCes't la vie! \ttres jolie"

# if you print 'la_vie'
print(la_vie)

## [1] "La vie en #FFC0CB (rose);\nCes't la vie! \ttres jolie"


# if you cat 'la_vie'
cat(la_vie)

## La vie en #FFC0CB (rose);
## Ces't la vie!  tres jolie
```

Here's what would happen to the string `la_vie` if we apply some substitutions with the POSIX character classes:

```
# remove space characters
gsub(pattern = "[[:blank:]]", replacement = "", la_vie)

## [1] "Lavieen#FFC0CB(rose);\nCes'tlavie!tresjolie"

# remove digits
gsub(pattern = "[[:punct:]]", replacement = "", la_vie)

## [1] "La vie en FFC0CB rose\nCest la vie \ttres jolie"

# remove digits
gsub(pattern = "[[:xdigit:]]", replacement = "", la_vie)

## [1] "L vi n # (ros);\ns't l vi! \ttrs joli"

# remove printable characters
gsub(pattern = "[[:print:]]", replacement = "", la_vie)

## [1] "\n\t"

# remove non-printable characters
gsub(pattern = "[^[:print:]]", replacement = "", la_vie)

## [1] "La vie en #FFC0CB (rose);Ces't la vie! tres jolie"

# remove graphical characters
gsub(pattern = "[[:graph:]]", replacement = "", la_vie)

## [1] "    \n    \t "

# remove non-graphical characters
gsub(pattern = "[^[:graph:]]", replacement = "", la_vie)

## [1] "Lavieen#FFC0CB(rose);Ces'tlavie!tresjolie"
```

5.2.6 Quantifiers

Another important set of regex elements are the so-called *quantifiers*. These are used when we want to match a **certain number** of characters that meet certain criteria.

Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found. The following table shows the regex quantifiers:

Quantifiers in R

Quantifier	Description
`?`	The preceding item is optional and will be matched at most once
`*`	The preceding item will be matched zero or more times
`+`	The preceding item will be matched one or more times
`{n}`	The preceding item is matched exactly `n` times
`{n,}`	The preceding item is matched `n` or more times
`{n,m}`	The preceding item is matched at least `n` times, but not more than `m` times

Some examples

```
# people names
people = c("rori", "emilia", "matteo", "mehmet", "filipe", "anna", "tyler",
    "rasmus", "jacob", "youna", "flora", "adi")

# match 'm' at most once
grep(pattern = "m?", people, value = TRUE)

##  [1] "rori"   "emilia" "matteo" "mehmet" "filipe" "anna"   "tyler"
##  [8] "rasmus" "jacob"  "youna"  "flora"  "adi"


# match 'm' exactly once
grep(pattern = "m{1}", people, value = TRUE, perl = FALSE)

## [1] "emilia" "matteo" "mehmet" "rasmus"


# match 'm' zero or more times, and 't'
grep(pattern = "m*t", people, value = TRUE)

## [1] "matteo" "mehmet" "tyler"


# match 't' zero or more times, and 'm'
grep(pattern = "t*m", people, value = TRUE)

## [1] "emilia" "matteo" "mehmet" "rasmus"


# match 'm' one or more times
grep(pattern = "m+", people, value = TRUE)
```

```
## [1] "emilia" "matteo" "mehmet" "rasmus"

# match 'm' one or more times, and 't'
grep(pattern = "m+.t", people, value = TRUE)

## [1] "matteo" "mehmet"

# match 't' exactly twice
grep(pattern = "t{2}", people, value = TRUE)

## [1] "matteo"
```

5.3 Functions for Regular Expressions

Once we've described how R handles some of the most common regular expression elements, it's time to present the functions we can use for working with regular expressions.

5.3.1 Main Regex functions

R contains a set of functions in the base package that we can use to find pattern matches. The following table lists these functions with a brief description:

Regular Expression Functions in R

Function	Purpose	Characteristic
`grep()`	finding regex matches	which elements are matched (index or value)
`grepl()`	finding regex matches	which elements are matched (TRUE & FALSE)
`regexpr()`	finding regex matches	positions of the first match
`gregexpr()`	finding regex matches	positions of all matches
`regexec()`	finding regex matches	hybrid of `regexpr()` and `gregexpr()`
`sub()`	replacing regex matches	only first match is replaced
`gsub()`	replacing regex matches	all matches are replaced
`strsplit()`	splitting regex matches	split vector according to matches

The first five functions listed in the previous table are used for finding pattern matches in character vectors. The goal is the same for all these functions: **finding a match**. The difference between them is in the format of the output. The next two functions —`sub()` and `gsub()`— are used for **substitution**: looking for matches with the purpose of replacing them. The last function, `strsplit()`, is used to **split** elements of a character vector into substrings according to regex matches.

Basically, all regex functions require two main arguments: a **pattern** (i.e. regular expression), and a **text** to match. Each function has other additional arguments but the main ones are a pattern and some text. In particular, the **pattern** is basically a character string containing a regular expression to be matched in the given **text**.

You can check the documentation of all the `grep()`-like functions by typing `help(grep)` (or alternatively `?grep`).

```
# help documentation for main regex functions
help(grep)
```

5.3.2 Regex functions in `stringr`

The R package `stringr` also provides several functions for regex operations (see table below). More specifically, `stringr` provides pattern matching functions to *detect*, *locate*, *extract*, *match*, *replace* and *split* strings.

Regex functions in `stringr`

Function	Description
`str_detect()`	Detect the presence or absence of a pattern in a string
`str_extract()`	Extract **first** piece of a string that matches a pattern
`str_extract_all()`	Extract **all** pieces of a string that match a pattern
`str_match()`	Extract **first** matched group from a string
`str_match_all()`	Extract **all** matched groups from a string
`str_locate()`	Locate the position of the **first** occurence of a pattern in a string
`str_locate_all()`	Locate the position of **all** occurences of a pattern in a string
`str_replace()`	Replace **first** occurrence of a matched pattern in a string
`str_replace_all()`	Replace **all** occurrences of a matched pattern in a string
`str_split()`	Split up a string into a variable number of pieces
`str_split_fixed()`	Split up a string into a fixed number of pieces

One of the important things to keep in mind is that all pattern matching functions in `stringr` have the following general form:

```
str_function(string, pattern)
```

The common characteristic is that they all share the first two arguments: a `string` vector to be processed and a single `pattern` (i.e. regular expression) to match.

5.3.3 Complementary matching functions

Together with the primary `grep()`-like functions, R has other related matching functions such as `regmatches()`, `match()`, `pmatch()`, `charmatch()`. The truth is that `regmatches()` is the only function that is designed to work with regex patterns. The other matching functions don't work with regular expressions but we can use them to match terms (e.g. words, argument names) within a character vector.

Complementary Matching Functions

Function	Purpose	Characteristic
`regmatches()`	extract or replace matches	use with data from `regexpr()`, `gregexpr()` or `regexec()`
`match()`	value matching	finding positions of (first) matches
`pmatch()`	partial string matching	finding positions
`charmatch()`	similar to `pmatch()`	finding positions

5.3.4 Accessory functions accepting regex patterns

Likewsie, R contains other functions that interact or accept regex patterns: `apropos()`, `ls()`, `browseEnv()`, `glob2rx()`, `help.search()`, `list.files()`. The main purpose of these functions is to search for R objects or files, but they can take a regex pattern as input.

Accessory Functions

Function	Description
`apropos()`	find objects by (partial) name
`browseEnv()`	browse objects in environment
`glob2rx()`	change wildcard or globbing pattern into Regular Expression
`help.search()`	search the help system
`list.files()`	list the files in a directory/folder

Chapter 6

Regular Expressions (part II)

In the previous chapter we talked about regular expressions in general; we discussed the particular way in which R works with regex patterns; and we also saw a quick presentation of the functions that we can use to manipulate strings with regular expressions. In this chapter we are going to describe in more detail the functions for regular expressions and we will see some examples that show their uses.

6.1 Pattern Finding Functions

Let's begin by reviewing the first five `grep()`-like functions `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, and `regexec()`. The goal is the same for all these functions: **finding a match**. The difference between them is in the format of the output. Essentially these functions require two main arguments: a `pattern` (i.e. regular expression), and a `text` to match. The basic usage for these functions is:

```
grep(pattern, text)
grepl(pattern, text)
regexpr(pattern, text)
gregexpr(pattern, text)
regexec(pattern, text)
```

Each function has other additional arguments but the important thing to keep in mind are a pattern and some text.

6.1.1 Function grep()

grep() is perhaps the most basic functions that allows us to match a pattern in a string vector. The first argument in grep() is a regular expression that specifies the pattern to match. The second argument is a character vector with the text strings on which to search. The output is the indices of the elements of the text vector for which there is a match. If no matches are found, the output is an empty integer vector.

```
# some text
text = c("one word", "a sentence", "you and me", "three two one")

# pattern
pat = "one"

# default usage
grep(pat, text)

## [1] 1 4
```

As you can see from the output in the previous example, grep() returns a numeric vector. This indicates that the 1st and 4th elements contained a match. In contrast, the 2nd and the 3rd elements did not.

We can use the argument value to modify the way in which the output is presented. If we choose value = TRUE, instead of returning the indices, grep() returns the content of the string vector:

```
# with 'value' (showing matched text)
grep(pat, text, value = TRUE)

## [1] "one word"      "three two one"
```

Another interesting argument to play with is invert. We can use this parameter to obtain unmatches strings by setting its value to TRUE

```
# with 'invert' (showing unmatched parts)
grep(pat, text, invert = TRUE)

## [1] 2 3

# same with 'values'
grep(pat, text, invert = TRUE, value = TRUE)

## [1] "a sentence" "you and me"
```

In summary, grep() can be used to subset a character vector to get only the elements containing (or not containing) the matched pattern.

6.1.2 Function `grepl()`

The function `grepl()` enables us to perform a similar task as `grep()`. The difference resides in that the output are not numeric indices, but logical (`TRUE` / `FALSE`). Hence you can think of `grepl()` as `grep`-*logical*. Using the same text string of the previous examples, here's the behavior of `grepl()`:

```
# some text
text = c("one word", "a sentence", "you and me", "three two one")

# pattern
pat = "one"

# default usage
grepl(pat, text)

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

Note that we get a logical vector of the same length as the character vector. Those elements that matched the pattern have a value of `TRUE`; those that didn't match the pattern have a value of `FALSE`.

6.1.3 Function `regexpr()`

To find exactly where the pattern is found in a given string, we can use the `regexpr()` function. This function returns more detailed information than `grep()` providing us:

a) which elements of the text vector actually contain the regex pattern, and

b) identifies the position of the substring that is matched by the regular expression pattern.

```
# some text
text = c("one word", "a sentence", "you and me", "three two one")

# default usage
regexpr("one", text)

## [1]  1 -1 -1 11      Starting Position of 1st match within  text [1:4]
## attr(,"match.length")
## [1]  3 -1 -1  3          “one” is three characters long
## attr(,"useBytes")
## [1] TRUE
```

At first glance the output from `regexpr()` may look a bit messy but it's very simple to interpret. What we have in the output are three displayed elements. The first element is an integer vector of the same length as text giving the starting positions of the first match. In this example the number 1 indicates that the pattern `"one"` starts at the position 1 of the first element in `text`. The negative index `-1` means that there was no match; the number 11 indicates the position of the substring that was matched in the fourth element of `text`.

The attribute `"match.length"` gives us the *length* of the match in each element of `text`. Again, a negative value of `-1` means that there was no match in that element. Finally, the attribute `"useBytes"` has a value of `TRUE` which means that the matching was done byte-by-byte rather than character-by-character.

6.1.4 Function `gregexpr()`

The function `gregexpr()` does practically the same thing as `regexpr()`: identify where a pattern is within a string vector, by searching each element separately. The only difference is that `gregexpr()` has an output in the form of a list. In other words, `gregexpr()` returns a list of the same length as `text`, each element of which is of the same form as the return value for `regexpr()`, except that the starting positions of every (disjoint) match are given.

```r
# some text
text = c("one word", "a sentence", "you and me", "three two one")

# pattern
pat = "one"

# default usage
gregexpr(pat, text)

## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
```

```
##
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] 11
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
```

6.1.5 Function `regexec()`

The function `regexec()` is very close to `gregexpr()` in the sense that the output is also a list of the same length as `text`. Each element of the list contains the starting position of the match. A value of `-1` reflects that there is no match. In addition, each element of the list has the attribute `"match.length"` giving the lengths of the matches (or -1 for no match):

```
# some text
text = c("one word", "a sentence", "you and me", "three two one")

# pattern
pat = "one"

# default usage
regexec(pat, text)
```

```
## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 3
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
```

```
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
## [[4]]
## [1] 11
## attr(,"match.length")
## [1] 3
```

Example from Spector

```
# handy function to extract matched term
x = regexpr(pat, text)
substring(text, x, x + attr(x, "match.length") - 1)

## [1] "one" ""    ""    "one"


# with NA
regexpr(pat, c(text, NA))

## [1]  1 -1 -1 11 NA
## attr(,"match.length")
## [1]  3 -1 -1  3 NA
```

6.2 Pattern Replacement Functions

Sometimes finding a pattern in a given string vector is all we want. However, there are occasions in which we might also be interested in *replacing* one pattern with another one. For this purpose we can use the substitution functions `sub()` and `gsub()`. The difference between `sub()` and `gsub()` is that the former replaces only the first occurrence of a pattern whereas the latter replaces all occurrences.

The replacement functions require three main arguments: a regex `pattern` to be matched, a `replacement` for the matched pattern, and the `text` where matches are sought. The basic usage is:

```
sub(pattern, replacement, text)
gsub(pattern, replacement, text)
```

6.2.1 Replacing first occurrence with sub()

The function `sub()` replaces the **first** occurrence of a pattern in a given text. This means that if there is more than one occurrence of the pattern in each element of a string vector, only the first one will be replaced. For example, suppose we have the following text vector containing various strings:

```
Rstring = c("The R Foundation",
            "for Statistical Computing",
            "R is FREE software",
            "R is a collaborative project")
```

Imagine that our aim is to replace the pattern `"R"` with a new pattern `"RR"`. If we use `sub()` this is what we obtain:

```
# string
Rstring = c("The R Foundation",
            "for Statistical Computing",
            "R is FREE software",
            "R is a collaborative project")

# substitute 'R' with 'RR'
sub("R", "RR", Rstring)

## [1] "The RR Foundation"             "for Statistical Computing"
## [3] "RR is FREE software"           "RR is a collaborative project"
```

As you can tell, only the first occurrence of the letter `R` is replaced in each element of the text vector. Note that the word `FREE` in the third element also contains an `R` but it was not replaced. This is because it was not the first occurrence of the pattern.

6.2.2 Replacing all occurrences with gsub()

To replace not only the first pattern occurrence, but **all** of the occurrences we should use `gsub()` (think of it as *general* substition). If we take the same vector `Rstring` and patterns of the last example, this is what we obtain when we apply `gsub()`

```
# string
Rstring = c("The R Foundation",
            "for Statistical Computing",
            "R is FREE software",
            "R is a collaborative project")
```

```
# substitute
gsub("R", "RR", Rstring)

## [1] "The RR Foundation"              "for Statistical Computing"
## [3] "RR is FRREE software"           "RR is a collaborative project"
```

The obtained output is almost the same as with `sub()`, except for the third element in `Rstring`. Now the occurence of `R` in the word `FREE` is taken into account and `gsub()` changes it to `FRREE`.

6.3 Splitting Character Vectors

Besides the operations of finding patterns and replacing patterns, another common task is *splitting* a string based on a pattern. To do this R comes with the function `strsplit()` which is designed to **split** the elements of a character vector into substrings according to regex matches.

If you check the help documentation —`help(strsplit)`— you will see that the basic usage of `strsplit()` requires two main arguments:

```
strsplit(x, split)
```

`x` is the character vector and `split` is the regular expression pattern. However, in order to keep the same notation that we've been using with the other `grep()` functions, it is better if we think of `x` as `text`, and `split` as `pattern`. In this way we can express the usage of `strsplit()` as:

```
strsplit(text, pattern)
```

One of the typical tasks in which we can use `strsplit()` is when we want to break a string into individual components (i.e. words). For instance, if we wish to separate each word within a given sentence, we can do that specifying a blank space `" "` as splitting pattern:

```
# a sentence
sentence = c("R is a collaborative project with many contributors")

# split into words
strsplit(sentence, " ")

## [[1]]
## [1] "R"             "is"            "a"             "collaborative"
## [5] "project"       "with"          "many"          "contributors"
```

Another basic example may consist in breaking apart the portions of a telephone number by splitting those sets of digits joined by a dash `"-"`

```
# telephone numbers
tels = c("510-548-2238", "707-231-2440", "650-752-1300")

# split each number into its portions
strsplit(tels, "-")
```

```
## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```

6.4 Functions in stringr

In the previous chapter we briefly presented the functions of the R package `stringr` for regular expressions. As we mentioned, all the `stringr` functions share a common usage structure:

```
 str_function(string, pattern)
```

The main two arguments are: a `string` vector to be processed , and a single `pattern` (i.e. regular expression) to match. Moreover, all the function names begin with the prefix `str_`, followed by the name of the action to be performed. For example, to *locate* the position of the first occurence, we should use `str_locate()`; to locate the positions of all matches we should use `str_locate_all()`.

6.4.1 Detecting patterns with str_detect()

For detecting whether a pattern is present (or absent) in a string vector, we can use the function `str_detect()`. Actually, this function is a wraper of `grepl()`:

```
# some objects
some_objs = c("pen", "pencil", "marker", "spray")

# detect phones
```

```
str_detect(some_objs, "pen")

## [1]  TRUE  TRUE FALSE FALSE

# select detected macthes
some_objs[str_detect(some_objs, "pen")]

## [1] "pen"    "pencil"
```

As you can see, the output of `str_detect()` is a boolean vector (`TRUE/FALSE`) of the same length as the specified `string`. You get a `TRUE` if a match is detected in a string, `FALSE` otherwise. Here's another more elaborated example in which the pattern matches dates of the form *day-month-year*:

```
# some strings
strings = c("12 Jun 2002", " 8 September 2004 ", "22-July-2009 ",
            "01 01 2001", "date", "02.06.2000",
            "xxx-yyy-zzzz", "$2,600")

# date pattern (month as text)
dates = "([0-9]{1,2})[- .]([a-zA-Z]+)[- .]([0-9]{4})"

# detect dates
str_detect(strings, dates)

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

6.4.2 Extract first match with `str_extract()`

For extracting a string containing a pattern, we can use the function `str_extract()`. In fact, this function extracts the first piece of a string that matches a given pattern. For example, imagine that we have a character vector with some tweets about *Paris*, and that we want to extract the hashtags. We can do this simply by defining a `#hashtag` pattern like `#[a-zA-Z]{1}`

```
# tweets about 'Paris'
paris_tweets = c(
  "#Paris is chock-full of cultural and culinary attractions",
  "Some time in #Paris along Canal St.-Martin famous by #Amelie",
  "While you're in #Paris, stop at cafe: http://goo.gl/yaCbW",
  "Paris, the city of light")

# hashtag pattern
```

```
hash = "#[a-zA-Z]{1,}"

# extract (first) hashtag
str_extract(paris_tweets, hash)

## [1] "#Paris" "#Paris" "#Paris" NA
```

As you can see, the output of `str_extract()` is a vector of same length as `string`. Those elements that don't match the pattern are indicated as `NA`. Note that `str_extract()` only matches the first pattern: it didn't extract the hashtag `"#Amelie"`.

6.4.3 Extract all matches with `str_extract_all()`

In addition to `str_extract()`, `stringr` also provides the function `str_extract_all()`. As its name indicates, we use `str_extract_all()` to extract **all** patterns in a vector string. Taking the same string as in the previous example, we can extract all the hashtag matches like so:

```
# extract (all) hashtags
str_extract_all(paris_tweets, "#[a-zA-Z]{1,}")

## [[1]]
## [1] "#Paris"
##
## [[2]]
## [1] "#Paris"  "#Amelie"
##
## [[3]]
## [1] "#Paris"
##
## [[4]]
## character(0)
```

Compared to `str_extract()`, the output of `str_extract_all()` is a **list** of same length as `string`. In addition, those elements that don't match the pattern are indicated with an empty character vector `character(0)` instead of `NA`.

6.4.4 Extract first match group with `str_match()`

Closely related to `str_extract()` the package `stringr` offers another extracting function: `str_match()`. This function not only extracts the matched pattern but it also shows each of the matched groups in a regex character class pattern.

```
# string vector
strings = c("12 Jun 2002", " 8 September 2004 ", "22-July-2009 ",
            "01 01 2001", "date", "02.06.2000",
            "xxx-yyy-zzzz", "$2,600")

# date pattern (month as text)
dates = "([0-9]{1,2})[- .]([a-zA-Z]+)[- .]([0-9]{4})"

# extract first matched group
str_match(strings, dates)
```

```
##      [,1]                [,2] [,3]        [,4]
## [1,] "12 Jun 2002"       "12" "Jun"       "2002"
## [2,] "8 September 2004" "8"  "September" "2004"
## [3,] "22-July-2009"      "22" "July"      "2009"
## [4,] NA                  NA   NA          NA
## [5,] NA                  NA   NA          NA
## [6,] NA                  NA   NA          NA
## [7,] NA                  NA   NA          NA
## [8,] NA                  NA   NA          NA
```

Note that the output is not a vector but a character matrix. The first column is the complete match, the other columns are each of the captured groups. For those unmatched elements, there is a missing value `NA`.

6.4.5 Extract all matched groups with `str_match_all()`

If what we're looking for is extracting **all** patterns in a string vector, instead of using `str_extract()` we should use `str_extract_all()`:

```
# tweets about 'Paris'
paris_tweets = c(
  "#Paris is chock-full of cultural and culinary attractions",
  "Some time in #Paris along Canal St.-Martin famous by #Amelie",
  "While you're in #Paris, stop at cafe: http://goo.gl/yaCbW",
  "Paris, the city of light")

# match (all) hashtags in 'paris_tweets'
str_match_all(paris_tweets, "#[a-zA-Z]{1,}")
```

```
## [[1]]
##       [,1]
## [1,] "#Paris"
##
## [[2]]
##       [,1]
## [1,] "#Paris"
## [2,] "#Amelie"
##
## [[3]]
##       [,1]
## [1,] "#Paris"
##
## [[4]]
## character(0)
```

Compared to `str_match()`, the output of `str_match_all()` is a **list**. Note al also that each element of the list is a matrix with as many rows as hashtag matches. In turn, those elements that don't match the pattern are indicated with an empty character vector `character(0)` instead of a `NA`.

6.4.6 Locate first match with `str_locate()`

Besides detecting, extracting and matching regex patterns, `stringr` allows us to *locate* occurences of patterns. For locating the position of the **first** occurence of a pattern in a string vector, we should use `str_locate()`.

```
# locate position of (first) hashtag
str_locate(paris_tweets, "#[a-zA-Z]{1,}")
```

```
##      start end
## [1,]     1   6
## [2,]    14  19
## [3,]    17  22
## [4,]    NA  NA
```

The output of `str_locate()` is a matrix with two columns and as many rows as elements in the (string) vector. The first column of the output is the `start` position, while the second column is the `end` position.

In the previous example, the result is a matrix with 4 rows and 2 columns. The first row corresponds to the hashtag of the first tweet. It starts at position 1 and ends at position 6. The second row corresponds to the hashtag of the second tweet; its start position is the

14th character, and its end position is the 19th character. The fourth row corresponds to the fourth tweet. Since there are no hashtags the values in that row are `NA`'s.

6.4.7 Locate all matches with `str_locate_all()`

To locate not just the first but **all** the occurence patterns in a string vector, we should use `str_locate_all()`:

```
# locate (all) hashtags in 'paris_tweets'
str_locate_all(paris_tweets, "#[a-zA-Z]{1,}")
```

```
## [[1]]
##      start end
## [1,]     1   6
##
## [[2]]
##      start end
## [1,]    14  19
## [2,]    54  60
##
## [[3]]
##      start end
## [1,]    17  22
##
## [[4]]
##      start end
```

Compared to `str_locate()`, the output of `str_locate_all()` is a **list** of the same length as the provided `string`. Each of the list elements is in turn a matrix with two columns. Those elements that don't match the pattern are indicated with an empty character vector instead of an `NA`.

Looking at the obtained result from applying `str_locate_all()` to `paris_tweets`, you can see that the second element contains the start and end positions for both hashtags `#Paris` and `#Amelie`. In turn, the fourth element appears empty since its associated tweet contains no hashtags.

6.4.8 Replace first match with `str_replace()`

For replacing the **first** occurrence of a matched pattern in a string, we can use `str_replace()`. Its usage has the following form:

```
str_replace(string, pattern, replacement)
```

In addition to the main 2 inputs of the rest of functions, `str_replace()` requires a third argument that indicates the **replacement** pattern.

Say we have the city names of San Francisco, Barcelona, Naples and Paris in a vector. And let's suppose that we want to replace the first vowel in each name with a semicolon. Here's how we can do that:

```
# city names
cities = c("San Francisco", "Barcelona", "Naples", "Paris")

# replace first matched vowel
str_replace(cities, "[aeiou]", ";")

## [1] "S;n Francisco" "B;rcelona"     "N;ples"        "P;ris"
```

Now, suppose that we want to replace the first consonant in each name. We just need to modify the `pattern` with a negated class:

```
# replace first matched consonant
str_replace(cities, "[^aeiou]", ";")

## [1] ";an Francisco" ";arcelona"     ";aples"        ";aris"
```

6.4.9 Replace all matches with `str_replace_all()`

For replacing **all** occurrences of a matched pattern in a string, we can use `str_replace_all()`. Once again, consider a vector with some city names, and let's suppose that we want to replace all the vowels in each name:

```
# city names
cities = c("San Francisco", "Barcelona", "Naples", "Paris")

# replace all matched vowel
str_replace_all(cities, pattern = "[aeiou]", ";")

## [1] "S;n Fr;nc;sc;" "B;rc;l;n;"     "N;pl;s"        "P;r;s"
```

Alternatively, to replace all consonants with a semicolon in each name, we just need to change the `pattern` with a negated class:

```
# replace all matched consonants
str_replace_all(cities, pattern = "[^aeiou]", ";")

## [1] ";a;;;;a;;i;;o" ";a;;e;o;a"     ";a;;e;"        ";a;i;"
```

6.4.10 String splitting with str_split()

Similar to `strsplit()`, `stringr` gives us the function `str_split()` to separate a character vector into a number of pieces. This function has the following usage:

```
str_split(string, pattern, n = Inf)
```

The argument `n` is the maximum number of pieces to return. The default value (`n = Inf`) implies that all possible split positions are used.

Let's see the same example of `strsplit()` in which we wish to split up a sentence into individuals words:

```
# a sentence
sentence = c("R is a collaborative project with many contributors")

# split into words
str_split(sentence, " ")

## [[1]]
## [1] "R"               "is"              "a"               "collaborative"
## [5] "project"         "with"            "many"            "contributors"
```

Likewise, we can break apart the portions of a telephone number by splitting those sets of digits joined by a dash `"-"`

```
# telephone numbers
tels = c("510-548-2238", "707-231-2440", "650-752-1300")

# split each number into its portions
str_split(tels, "-")

## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```

The result is a **list** of character vectors. Each element of the string vector corresponds to an element in the resulting list. In turn, each of the list elements will contain the split vectors (i.e. number of pieces) occurring from the matches.

In order to show the use of the argument `n`, let's consider a vector with flavors `"chocolate"`, `"vanilla"`, `"cinnamon"`, `"mint"`, and `"lemon"`. Suppose we want to split each flavor name defining as pattern the class of vowels:

```
# string
flavors = c("chocolate", "vanilla", "cinnamon", "mint", "lemon")

# split by vowels
str_split(flavors, "[aeiou]")
```

```
## [[1]]
## [1] "ch" "c"  "l"  "t"  ""
##
## [[2]]
## [1] "v"  "n"  "ll" ""
##
## [[3]]
## [1] "c"  "nn" "m"  "n"
##
## [[4]]
## [1] "m"  "nt"
##
## [[5]]
## [1] "l" "m" "n"
```

Now let's modify the maximum number of pieces to `n = 2`. This means that `str_split()` will split each element into a maximum of 2 pieces. Here's what we obtain:

```
# split by first vowel
str_split(flavors, "[aeiou]", n = 2)
```

```
## [[1]]
## [1] "ch"     "colate"
##
## [[2]]
## [1] "v"     "nilla"
##
## [[3]]
## [1] "c"      "nnamon"
##
## [[4]]
## [1] "m"  "nt"
##
## [[5]]
## [1] "l"   "mon"
```

6.4.11 String splitting with `str_split_fixed()`

In addition to `str_split()`, there is also the `str_split_fixed()` function that splits up a string into a fixed number of pieces. Its usage has the following form:

```
str_split_fixed(string, pattern, n)
```

Note that the argument `n` does not have a default value. In other words, we need to specify an integer to indicate the number of pieces.

Consider again the same vector of `flavors`, and the letter `"n"` as the pattern to match. Let's see the behavior of `str_split_fixed()` with `n = 2`.

```
# string
flavors = c("chocolate", "vanilla", "cinnamon", "mint", "lemon")

# split flavors into 2 pieces
str_split_fixed(flavors, "n", 2)

##      [,1]        [,2]
## [1,] "chocolate" ""
## [2,] "va"        "illa"
## [3,] "ci"        "namon"
## [4,] "mi"        "t"
## [5,] "lemo"      ""
```

As you can tell, the output is a character matrix with as many columns as `n = 2`. Since `"chocolate"` does not contain any letter `"n"`, its corresponding value in the second column remains empty `""`. In contrast, the value of the second column associated to `"lemon"` is also empty. But this is because this flavor is split up into `"lemo"` and `""`.

If we change the value `n = 3`, we will obtain a matrix with three columns:

```
# split favors into 3 pieces
str_split_fixed(flavors, "n", 3)

##      [,1]        [,2]   [,3]
## [1,] "chocolate" ""     ""
## [2,] "va"        "illa" ""
## [3,] "ci"        ""     "amon"
## [4,] "mi"        "t"    ""
## [5,] "lemo"      ""     ""
```

Chapter 7

Practical Applications

This chapter is dedicated to show some practical examples that involve handling and processing strings in R. The main idea is to put in practice all the material covered so far and get a grasp of the variety of things we can do in R.

We will describe four typical examples. The examples are not exhaustive but just representative:

1. reversing a string
2. matching email addresses
3. matching html elements (`href`'s and `img`'s anchors)
4. some stats and analytics of character data

7.1 Reversing a string

Our first example has to do with reversing a character string. More precisely, the objective is to create a function that takes a string and returns it in reversed order. The trick of this exercise depends on what we understand with the term *reversing*. For some people, reversing may be understood as simply having the set of *characters* in reverse order. For other people instead, reversing may be understood as having a set of *words* in reverse order. Can you see the distinction?

Let's consider the following two simple strings:

- `"atmosphere"`
- `"the big bang theory"`

The first string is formed by one single word (atmosphere). The second string is formed by a sentence with four words (the big bang theory). If we were to reverse both strings by characters we would get the following results:

- `"erehpsomta"`
- `"yroeht gnab gib eht"`

Conversely, if we were to reverse the strings by words, we would obtain the following output:

- `"atmosphere"`
- `"theory bang big the"`

For this example we will implement a function for each type of reversing operation.

7.1.1 Reversing a string by characters

The first case for reversing a string is to do it by **characters**. This implies that we need to split a given string into its different characters, and then we need to concatenate them back together in reverse order. Let's try to write a first function:

```r
# function that reverses a string by characters
reverse_chars <- function(string)
{
  # split string by characters
  string_split = strsplit(string, split = "")
  # reverse order
  rev_order = nchar(string):1
  # reversed characters
  reversed_chars = string_split[[1]][rev_order]
  # collapse reversed characters
  paste(reversed_chars, collapse="")
}
```

Let's test our reversing function with a character and numeric vectors:

```r
# try 'reverse_chars'
reverse_chars("abcdefg")

## [1] "gfedcba"

# try with non-character input
reverse_chars(12345)

## Error:  non-character argument
```

As you can see, `reverse_chars()` works fine when the input is in `"character"` mode. However, it complains when the input is `"non-character"`. In order to make our function more robust, we can force the input to be converted as character. The resulting code is given as:

```
# reversing a string by characters
reverse_chars <- function(string)
{
  string_split = strsplit(as.character(string), split = "")
  reversed_split = string_split[[1]][nchar(string):1]
  paste(reversed_split, collapse="")
}
```

Now if we try our modified function, we get the expected results:

```
# example with one word
reverse_chars("atmosphere")

## [1] "erehpsomta"

# example with a several words
reverse_chars("the big bang theory")

## [1] "yroeht gnab gib eht"
```

Moreover, it also works with non-character input:

```
# try 'reverse_chars'
reverse_chars("abcdefg")

## [1] "gfedcba"

# try with non-character input
reverse_chars(12345)

## [1] "54321"
```

If we want to use our function with a vector (more than one element), we can combine it with the `lapply()` function as follows:

```
# reverse vector (by characters)
lapply(c("the big bang theory", "atmosphere"), reverse_chars)

## [[1]]
## [1] "yroeht gnab gib eht"
##
## [[2]]
## [1] "erehpsomta"
```

7.1.2 Reversing a string by words

The second type of reversing operation is to reverse a string by **words**. In this case the procedure involves splitting up a string by words, re-arrange them in reverse order, and paste them back in one sentence. Here's how we can defined our `reverse_words()` function:

```
# function that reverses a string by words
reverse_words <- function(string)
{
  # split string by blank spaces
  string_split = strsplit(as.character(string), split = " ")
  # how many split terms?
  string_length = length(string_split[[1]])
  # decide what to do
  if (string_length == 1) {
    # one word (do nothing)
    reversed_string = string_split[[1]]
  } else {
    # more than one word (collapse them)
    reversed_split = string_split[[1]][string_length:1]
    reversed_string = paste(reversed_split, collapse = " ")
  }
  # output
  return(reversed_string)
}
```

The first step inside `reverse_words()` is to split the `string` according to a blank space pattern `" "`. Then we are counting the number of components resulting from the splitting step. Based on this information there are two options. If there is only one word, then there is nothing to do. If we have more than one words, then we need to re-arrenge them in reverse order and collapse them in a single string.

Once we have defined our function, we can try it on the two string examples to check that it works as expected:

```
# examples
reverse_words("atmosphere")

## [1] "atmosphere"

reverse_words("the big bang theory")

## [1] "theory bang big the"
```

Similarly, to use our function on a vector with more than one element, we should call it within the `lapply()` function as follows:

```
# reverse vector (by words)
lapply(c("the big bang theory", "atmosphere"), reverse_words)

## [[1]]
## [1] "theory bang big the"
##
## [[2]]
## [1] "atmosphere"
```

7.2 Matching e-mail addresses

The second practical example that we will discuss consists of matching an email address. We will work with usual email addresses having one (or a similar variant) of the following forms:

```
somename@email.com
somename99@email.com
some.name@email.com
some.name@an-email.com
some.name@an.email.com
```

Since our goal is to match an email address, this implies that we need to define a corresponding regex pattern. If we look at the previous email forms it is possible to see that they have a general structure that can be broken into three parts. The first part is the username (e.g. `somename99`). The second part is an @ symbol. The third part is the domain name (e.g. `an.email.com`).

The username pattern can be defined as:

```
^([a-z0-9_\\.-]+)
```

The username pattern starts with a caret ^ to indicate the beginning of the string. Then we have a group indicated with parentheses. It matches one or more lowercase letters, numbers, underscores, dots, or hyphens.

The domain name pattern can be defined as:

```
([\\da-z\\.-]+)\\.([a-z\\.]{2,6})$
```

The domain name should be one or more lowercase letters, numbers, underscores, dots, or hyphens. Then another (escaped) dot, followed by an extension of two to six letters or dots. And finally the end of the string ($).

The complete regular expression pattern (in R) for an email address is:

```
"^([a-z0-9_\\.-]+)@([\\da-z\\.-]+)\\.([a-z\\.]{2,6})$"
```

Let's test our pattern with a minimalist example:

```
# pattern
email_pat = "^([a-z0-9_\\.-]+)@([\\da-z\\.-]+)\\.([a-z\\.]{2,6})$"

# string that matches
grepl(pattern = email_pat, x = "gaston@abc.com")

## [1] TRUE
```

Here is another more real example:

```
# another string that matches
grep(pattern = email_pat, x = "gaston.sanchez@research-center.fr")

## [1] 1
```

However, if we have a "long" TLD (top-level domain) exceeding six letters, the pattern won't match, like in the next example:

```
# unmatched email (TLD too long)
grep(pattern = email_pat, x = "gaston@abc.something")

## integer(0)
```

Now let's apply the email pattern to test whether several strings match or not:

```
# potential email addresses
emails = c(
 "simple@example.com",
 "johnsmith@email.gov",
 "marie.curie@college.edu",
 "very.common@example.com",
 "a.little.lengthy.but.ok@dept.example.com",
 "disposable.style.email.with+symbol@example.com",
 "not_good@email.address")

# detect pattern
str_detect(string=emails, pattern=email_pat)

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

Note that the two last elements in `emails` are not well defined email addresses (they don't match the espicified pattern). The fifth address contains five sets of strings including a

`+` symbol. In turn, the sixth address has a long domain name (`email.address`) in which `address` exceeds six letters.

7.3 Matching HTML elements

For our third example we will deal with some basic handling of HTML tags. We'll take the webpage for the R mailing lists: http://www.r-project.org/mail.html

If you visit the previous webpage you will see that there are four general mailing lists devoted to R:

- **R-announce** is where major announcements about the development of R and the availability of new code.
- **R-packages** is a list of announcements on the availability of new or enhanced contributed packages
- **R-help** is the main R mailing list for discussion about problems and solutions using R
- **R-devel** is a list intended for questions and discussion about code development in R

Additionally, there are several specific **Special Interest Group** (SIG) mailing lists. The following table shows the first 5 groups:

First 5 Special Interest Groups (SIG) in R

Name	Description
`R-SIG-Mac`	Special Interest Group on Mac ports of R
`R-sig-DB`	SIG on Database Interfaces
`R-SIG-Debian`	Special Interest Group for Debian ports of R
`R-sig-dynamic-models`	Special Interest Group for Dynamic Simulation Models
`R-sig-Epi`	R for epidemiological data analysis

As a simple example, suppose we wanted to get the `href` attributes of all the SIG links. For instance, the `href` attribute of the R-SIG-Mac link is:

```
https://stat.ethz.ch/mailman/listinfo/r-sig-mac
```

In turn the `href` attribute of the R-sig-DB link is:

```
https://stat.ethz.ch/mailman/listinfo/r-sig-db
```

If we take a peek at the html source-code of the webpage, we'll see that all the links can be found on lines like this one:

```
    <td><a href="https://stat.ethz.ch/mailman/listinfo/r-sig-mac">
<tt>R-SIG-Mac</tt></a></td>
```

7.3.1 Getting SIG links

The first step is to create a vector of character strings that will contain the lines of the mailing lists webpage. We can create this vector by simply passing the URL name to `readLines()`:

```
# read html content
mail_lists = readLines("http://www.r-project.org/mail.html")
```

Once we've read the HTML content of the R mailing lists webpage, the next step is to define our regex pattern that matches the SIG links.

```
'^.*<td> *<a href="(https.*)">.*$'
```

Let's examine the proposed pattern. By using the caret (^) and dollar sign ($) we can describe our pattern as an entire line. Next to the caret we match anything zero or more times followed by a `<td>` tag. Then there is a blank space matched zero or more times, followed by an anchor tag with its `href` attribute. Note that we are using double quotation marks to match the `href` attribute (`"(https.*)"`). Moreover, the entire regex pattern is surrounded by single quotations marks `' '`. Here is how we can get the SIG links:

```
# SIG's href pattern
sig_pattern = '^.*<td> *<a href="(https.*)">.*$'

# find SIG href attributes
sig_hrefs = grep(sig_pattern, mail_lists, value = TRUE)

# let's see first 5 elements (shorten output)
shorten_sigs = rep("", 5)
for (i in 1:5) {
  shorten_sigs[i] = toString(sig_hrefs[i], width=70)
}
shorten_sigs

## [1] "    <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-mac\">...."
## [2] "    <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-db\"><...."
## [3] "    <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-debia...."
## [4] "    <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-dynam...."
## [5] "    <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-epi\">...."
```

We need to get rid of the extra html tags. We can easily extract the names of the note files using the `sub()` function (since there is only one link per line, we don't need to use `gsub()`, although we could).

```
# get first matched group
sub(sig_pattern, "\\1", sig_hrefs)
```

```
##  [1] "https://stat.ethz.ch/mailman/listinfo/r-sig-mac"
##  [2] "https://stat.ethz.ch/mailman/listinfo/r-sig-db"
##  [3] "https://stat.ethz.ch/mailman/listinfo/r-sig-debian"
##  [4] "https://stat.ethz.ch/mailman/listinfo/r-sig-dynamic-models"
##  [5] "https://stat.ethz.ch/mailman/listinfo/r-sig-epi"
##  [6] "https://stat.ethz.ch/mailman/listinfo/r-sig-ecology"
##  [7] "https://stat.ethz.ch/mailman/listinfo/r-sig-fedora"
##  [8] "https://stat.ethz.ch/mailman/listinfo/r-sig-finance"
##  [9] "https://stat.ethz.ch/mailman/listinfo/r-sig-geo"
## [10] "https://stat.ethz.ch/mailman/listinfo/r-sig-gr"
## [11] "https://stat.ethz.ch/mailman/listinfo/r-sig-gui"
## [12] "https://stat.ethz.ch/mailman/listinfo/r-sig-hpc"
## [13] "https://stat.ethz.ch/mailman/listinfo/r-sig-jobs"
## [14] "https://stat.ethz.ch/mailman/listinfo/r-sig-mixed-models"
## [15] "https://stat.ethz.ch/mailman/listinfo/r-sig-mediawiki"
## [16] "https://stat.ethz.ch/mailman/listinfo/r-sig-networks"
## [17] "https://stat.ethz.ch/mailman/listinfo/r-sig-phylo"
## [18] "https://stat.ethz.ch/mailman/listinfo/r-sig-qa"
## [19] "https://stat.ethz.ch/mailman/listinfo/r-sig-robust"
## [20] "https://stat.ethz.ch/mailman/listinfo/r-sig-s"
## [21] "https://stat.ethz.ch/mailman/listinfo/r-sig-teaching"
## [22] "https://stat.ethz.ch/mailman/listinfo/r-sig-wiki"
```

As you can see, we are using the regex pattern \\1 in the `sub()` function. Generally speaking `\\N` is replaced with the `N`-th group specified in the regular expression. The first matched group is referenced by \\1. In our example, the first group is everything that is contained in the curved brackets, that is: `(https.*)`, which are in fact the links we are looking for.

7.4 Text Analysis of BioMed Central Journals

For our last application we will work analyzing some text data. We will analyze the catalog of journals from the **BioMed Central** (BMC), a scientific publisher that specializes in open access journal publication. You can find more informaiton of BMC at: http://www.biomedcentral.com/about/catalog

The data with the journal catalog is available in csv format at: http://www.biomedcentral.com/journals/biomedcentraljournallist.txt

To import the data in R we can read the file with `read.table()`. Just specify the URL location of the file and pass it to `read.table()`. Don't forget to set the arguments `sep = ","` and `stringsAsFactors = FALSE`

```
# link of data set
url = "http://www.biomedcentral.com/journals/biomedcentraljournallist.txt"

# read data (stringsAsFactors=FALSE)
biomed = read.table(url, header = TRUE, sep = ",", stringsAsFactors = FALSE)
```

We can check the structure of the data with the function `str()`:

```
# structure of the dataset
str(biomed, vec.len = 1)

## 'data.frame': 336 obs. of  7 variables:
##  $ Publisher     : chr  "BioMed Central Ltd" ...
##  $ Journal.name  : chr  "AIDS Research and Therapy" ...
##  $ Abbreviation  : chr  "AIDS Res Ther" ...
##  $ ISSN          : chr  "1742-6405" ...
##  $ URL           : chr  "http://www.aidsrestherapy.com" ...
##  $ Start.Date    : int  2004 2011 ...
##  $ Citation.Style: chr  "BIOMEDCENTRAL" ...
```

As you can see, the data frame `biomed` has 336 observations and 7 variables. Actually, all the variables except for `Start.Date` are in character mode.

7.4.1 Analyzing Journal Names

We will do a simple analysis of the journal names. The goal is to study what are the more common terms used in the title of the journals. We are going to keep things at a basic level but for a more formal (and sophisticated) analysis you can check the package `tm` —text mining— (by Ingo Feinerer).

To have a better idea of what the data looks like, let's check the first journal names.

```
# first 5 journal names
head(biomed$Journal.name, 5)

## [1] "AIDS Research and Therapy"
## [2] "AMB Express"
## [3] "Acta Neuropathologica Communications"
## [4] "Acta Veterinaria Scandinavica"
## [5] "Addiction Science & Clinical Practice"
```

As you can tell, the fifth journal `"Addiction Science & Clinical Practice"` has an ampersand `&` symbol. Whether to keep the ampersand and other punctutation symbols depends on the objectives of the analysis. In our case, we will remove those elements.

Preprocessing

The preprocessing steps implies to get rid of the punctuation symbols. For convenient reasons it is always recommended to start working with a small subset of the data. In this way we can experiment at a small scale until we are confident with the right manipulations. Let's take the first 10 journals:

```
# get first 10 names
titles10 = biomed$Journal.name[1:10]
titles10

##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
##  [3] "Acta Neuropathologica Communications"
##  [4] "Acta Veterinaria Scandinavica"
##  [5] "Addiction Science & Clinical Practice"
##  [6] "Agriculture & Food Security"
##  [7] "Algorithms for Molecular Biology"
##  [8] "Allergy, Asthma & Clinical Immunology"
##  [9] "Alzheimer's Research & Therapy"
## [10] "Animal Biotelemetry"
```

We want to get rid of the ampersand signs `&`, as well as other punctuation marks. This can be done with `str_replace_all()` and replacing the pattern `[[:punct:]]` with empty strings `""` (don't forget to load the `stringr` package)

```
# remove punctuation
titles10 = str_replace_all(titles10, pattern = "[[:punct:]]", "")
titles10

##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
##  [3] "Acta Neuropathologica Communications"
##  [4] "Acta Veterinaria Scandinavica"
##  [5] "Addiction Science  Clinical Practice"
##  [6] "Agriculture  Food Security"
##  [7] "Algorithms for Molecular Biology"
##  [8] "Allergy Asthma  Clinical Immunology"
##  [9] "Alzheimers Research  Therapy"
## [10] "Animal Biotelemetry"
```

We succesfully replaced the punctuation symbols with empty strings, but now we have extra whitespaces. To remove the whitespaces we will use again `str_replace_all()` to replace any one or more whitespaces \\`s+` with a single blank space `" "`.

```
# trim extra whitespaces
titles10 = str_replace_all(titles10, pattern = "\\s+", " ")
titles10
```

Nice!

```
##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
##  [3] "Acta Neuropathologica Communications"
##  [4] "Acta Veterinaria Scandinavica"
##  [5] "Addiction Science Clinical Practice"
##  [6] "Agriculture Food Security"
##  [7] "Algorithms for Molecular Biology"
##  [8] "Allergy Asthma Clinical Immunology"
##  [9] "Alzheimers Research Therapy"
## [10] "Animal Biotelemetry"
```

Once we have a better idea of how to preprocess the journal names, we can proceed with all the 336 titles.

```
# remove punctuation symbols
all_titles = str_replace_all(biomed$Journal.name, pattern = "[[:punct:]]", "")

# trim extra whitespaces
all_titles = str_replace_all(all_titles, pattern = "\\s+", " ")
```

The next step is to split up the titles into its different terms (the output is a list).

```
# split titles by words
all_titles_list = str_split(all_titles, pattern = " ")

# show first 2 elements
all_titles_list[1:2]

## [[1]]
## [1] "AIDS"     "Research" "and"      "Therapy"
##
## [[2]]
## [1] "AMB"     "Express"
```

Summary statistics

So far we have a list that contains the words of each journal name. Wouldn't be interesting to know more about the distribution of the number of terms in each title? This means that we need to calculate how many words are in each title. To get these numbers let's use `length()` within `sapply()`; and then let's tabulate the obtained frequencies:

```
# how many words per title
words_per_title = sapply(all_titles_list, length)

# table of frequencies
table(words_per_title)

## words_per_title
##   1   2   3   4   5   6   7   8   9
##  17 108  81  55  33  31   6   4   1
```

We can also express the distribution as percentages, and we can get some summary statistics with `summary()`

```
# distribution
100 * round(table(words_per_title)/length(words_per_title), 4)

## words_per_title
##     1     2     3     4     5     6     7     8     9
##  5.06 32.14 24.11 16.37  9.82  9.23  1.79  1.19  0.30


# summary
summary(words_per_title)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.00    3.00    3.36    4.00    9.00
```

Looking at summary statistics we can say that around 30% of journal names have 2 words. Likewise, the median number of words per title is 3 words.

Interestingly the maximum value is 9 words. What is the journal with 9 terms in its title? We can find the longest journal name as follows:

```
# longest journal
all_titles[which(words_per_title == 9)]

## [1] "Journal of Venomous Animals and Toxins including Tropical Diseases"
```

7.4.2 Common words

Remember that our main goal with this example is to find out what words are the most common in the journal titles. To answer this question we first need to create something like a *dictionary* of words. How do get such dictionary? Easy, we just have to obtain a vector containing all the words in the titles:

```
# vector of words in titles
title_words = unlist(all_titles_list)

# get unique words
unique_words = unique(title_words)

# how many unique words in total
num_unique_words = length(unique(title_words))
num_unique_words

## [1] 441
```

Applying `unique()` to the vector `title_words` we get the desired dictionary of terms, which has a total of 441 words.

Once we have the unique words, we need to count how many times each of them appears in the titles. Here's a way to do that:

```
# vector to store counts
count_words = rep(0, num_unique_words)

# count number of occurrences
for (i in 1:num_unique_words) {
    count_words[i] = sum(title_words == unique_words[i])
}
```

An alternative simpler way to count the number of word occurrences is by using the `table()` function on `title_words`:

```
# table with word frequencies
count_words_alt = table(title_words)
```

In any of both cases (`count_words` or `count_words_alt`), we can examine the obtained frequencies with a simple table:

```
# table of frequencies
table(count_words)

## count_words
##   1   2   3   4   5   6   7   9  10  12  13  16  22  24  28  30  65  67
## 318  61  24   8   5   7   1   2   2   2   1   2   1   1   1   1   1   1
##  83  86
##   1   1


# equivalently
table(count_words_alt)

## count_words_alt
##   1   2   3   4   5   6   7   9  10  12  13  16  22  24  28  30  65  67
## 318  61  24   8   5   7   1   2   2   2   1   2   1   1   1   1   1   1
##  83  86
##   1   1
```

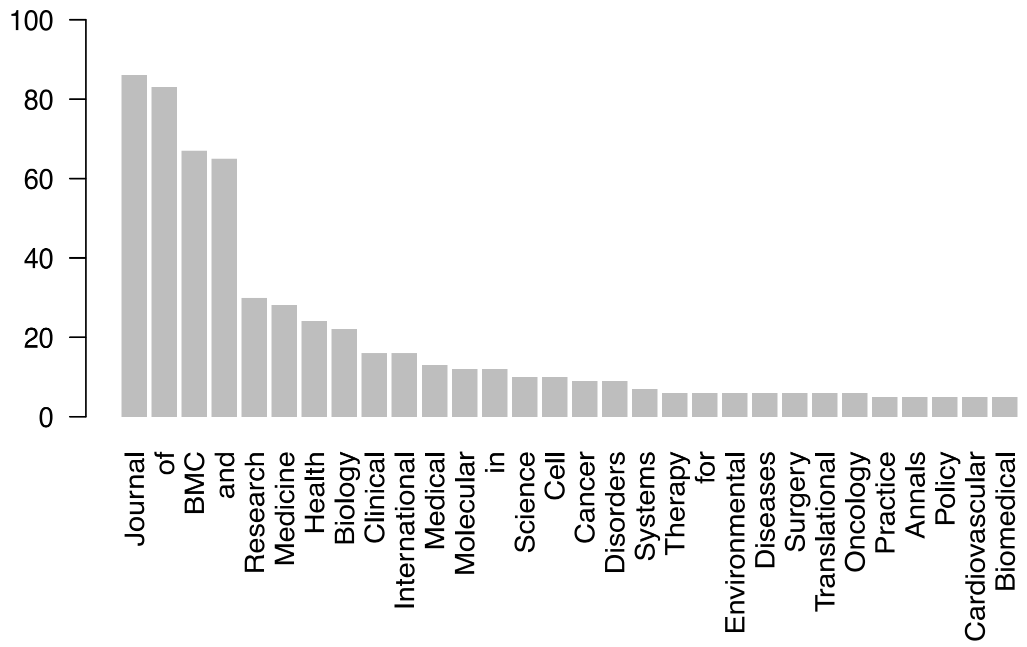
The top 30 words

For illustration purposes let's examine which are the top 30 common words.

```
# index values in decreasing order
top_30_order = order(count_words, decreasing = TRUE)[1:30]

# top 30 frequencies
top_30_freqs = sort(count_words, decreasing = TRUE)[1:30]

# select top 30 words
top_30_words = unique_words[top_30_order]
top_30_words

##  [1] "Journal"        "of"             "BMC"            "and"
##  [5] "Research"       "Medicine"       "Health"         "Biology"
##  [9] "Clinical"       "International"  "Medical"        "Molecular"
## [13] "in"             "Science"        "Cell"           "Cancer"
## [17] "Disorders"      "Systems"        "Therapy"        "for"
## [21] "Environmental"  "Diseases"       "Surgery"        "Translational"
## [25] "Oncology"       "Practice"       "Annals"         "Policy"
## [29] "Cardiovascular" "Biomedical"
```

To visualize the `top_30_words` we can plot them with a barchart using `barplot()`:

```
# barplot
barplot(top_30_freqs, border = NA, names.arg = top_30_words,
        las = 2, ylim = c(0,100))
```



Wordcloud

To finish this section let's try another visualization output by using a *wordcloud*, also known as tag cloud. To get this type of graphical display we'll use the package `wordcloud` (by Ian Fellows). If you haven't downloaded the package remember to install it with `install.packages()`

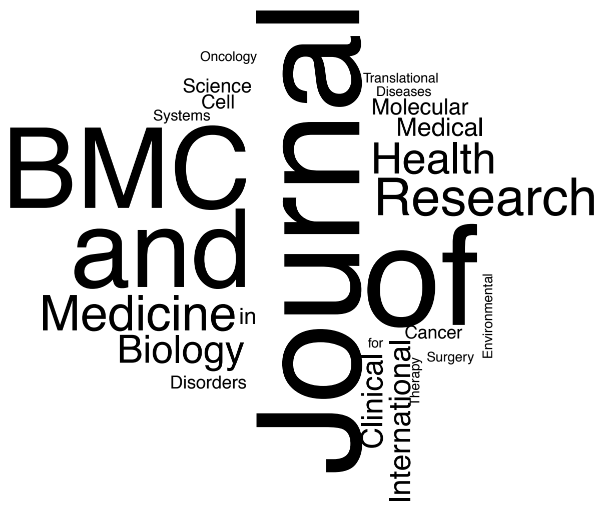
```r
# installing wordcloud
install.packages("wordcloud")

# load wordcloud
library(wordcloud)
```

To plot a tag cloud with just need to use the function `wordcloud()`. The most basic usage of this function has the following form:

```
wordcloud(words, freq}
```

It requires two main arguments: a character vector of `words` and a numeric vector `freq` with the frequency of the words. In our example, the vector of `words` corresponds to the vector `unique_words`. In turn, the vector of frequencies corresponds to `count_words`. Here's the code to plot the wordcloud with those terms having a minimun frequency of 6.

```r
# wordcloud
wordcloud(unique_words, count_words, scale=c(8,.2), min.freq=6,
          max.words=Inf, random.order=FALSE, rot.per=.15)
```


Journal
BMC
and
of
Research
Health
Medicine
Biology
Medical
Molecular
International
Clinical
Cancer
Science
Cell
Disorders
Systems
Oncology
Translational
Diseases
in
for
Surgery
Therapy
Environmental