

R for Strings

Handling Strings With R

Gaston Sanchez

Table of contents

About	7
My Series of R Tutorials	8
Donation	8
License	9
I I Characters in R	10
1 Introductory Appetizer	11
1.1 A Toy Example	11
1.1.1 Abbreviating strings	11
1.1.2 Getting the longest name	13
1.1.3 Selecting States	14
1.1.4 Some computations	15
2 Character Strings in R	19
2.1 Characters in R	19
2.2 Getting Started with Strings	20
2.3 Creating Character Strings	21
2.3.1 Empty string	23
2.3.2 Empty character vector	23
2.3.3 Function <code>c()</code>	25
2.3.4 <code>is.character()</code> and <code>as.character()</code>	25
2.4 Strings and R Objects	26
2.4.1 Behavior of R objects with character strings	27
2.5 The Workhorse Function <code>paste()</code>	30
2.6 Getting Text into R	32
2.6.1 Reading tables	32
2.6.2 Reading raw text	34
3 Basic Manipulations with "base" Functions	37
3.1 Basic String Manipulations	37
3.1.1 Count number of characters with <code>nchar()</code>	37
3.1.2 Convert to lower case with <code>tolower()</code>	38
3.1.3 Convert to upper case with <code>toupper()</code>	39

3.1.4	Upper or lower case conversion with <code>casefold()</code>	39
3.1.5	Character translation with <code>chartr()</code>	40
3.1.6	Abbreviate strings with <code>abbreviate()</code>	41
3.1.7	Replace substrings with <code>substr()</code>	43
3.1.8	Replace substrings with <code>substring()</code>	45
3.2	Set Operations	46
3.2.1	Set union with <code>union()</code>	46
3.2.2	Set intersection with <code>intersect()</code>	46
3.2.3	Set difference with <code>setdiff()</code>	47
3.2.4	Set equality with <code>setequal()</code>	47
3.2.5	Exact equality with <code>identical()</code>	48
3.2.6	Element contained with <code>is.element()</code>	48
3.2.7	Sorting with <code>sort()</code>	49
3.2.8	Repetition with <code>paste(rep())</code>	50
4	Basic Manipulations with "stringr" Functions	51
4.1	Package "stringr"	52
4.2	Basic String Operations	52
4.2.1	Concatenating with <code>str_c()</code>	53
4.2.2	Number of characters with <code>str_length()</code>	54
4.2.3	Substring with <code>str_sub()</code>	55
4.2.4	Duplication with <code>str_dup()</code>	58
4.2.5	Padding with <code>str_pad()</code>	59
4.2.6	Wrapping with <code>str_wrap()</code>	60
4.2.7	Trimming with <code>str_trim()</code>	61
4.2.8	Word extraction with <code>word()</code>	62
II	II Print & Format	64
5	Formatting Text and Numbers	65
5.1	Printing Characters	65
5.1.1	Generic printing with <code>print()</code>	65
5.1.2	Unquoted characters with <code>noquote()</code>	67
5.1.3	Concatenate and print with <code>cat()</code>	68
5.1.4	Encoding strings with <code>format()</code>	70
6	C-style Formatting	73
6.1	C-style Formatting Options	74
6.1.1	Format Slot Syntax	74
6.1.2	Example: basic <code>sprintf()</code>	75
6.1.3	Example: File Names	77
6.1.4	Example: Fahrenheit to Celsius	78

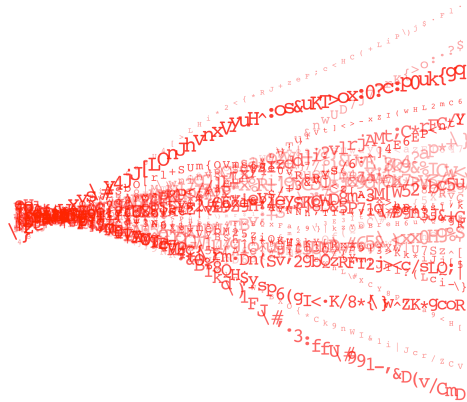
6.1.5	Example: Car Traveled Distance	79
6.1.6	Example: Coffee Prices	81
6.1.7	Converting objects to strings with <code>toString()</code>	84
6.1.8	Comparing printing methods	85
7	Input and Output	87
7.1	Output	87
7.1.1	Concatenating output	87
7.1.2	Sinking output	89
7.2	Exporting Tables	90
III	III Regex	91
8	Getting Started with Regular Expressions	92
8.1	What are Regular Expressions?	92
8.1.1	What are Regular Expressions used for?	93
8.1.2	A word of caution about regex	93
8.1.3	About Regular Expressions in R	93
8.2	Regex Basics	94
8.3	Literal Characters	95
8.3.1	Matching Literal Characters	95
8.4	R Functions for Regular Expressions	96
8.4.1	Regex Functions in "base" Package	96
8.4.2	Regex Functions in Package "stringr"	97
8.4.3	Matching Literal Characters With "stringr" Functions	98
8.5	Metacharacters	99
8.5.1	About Metacharacters	99
8.6	The Wildcard Metacharacter	100
8.6.1	Escaping metacharacters	102
9	Character Sets	105
9.1	Defining character sets	105
9.2	Character ranges	106
9.3	Negative Character Sets	108
9.4	Metacharacters Inside Character Sets	109
9.5	Character Classes	111
9.6	POSIX Character Classes	114
10	Anchors	117
10.1	Anchors	117
10.1.1	Start of String	118
10.1.2	End of String	118

11	Quantifiers	122
11.1	Quantifier Metacharacters	122
11.1.1	What do groups mean in Regex?	124
11.2	Greedy vs Lazy Match	125
12	Boundaries and Look Arouds	127
12.1	Boundaries	127
12.2	Look Arouds	130
12.2.1	Look Aheads	130
12.2.2	Look Behinds	133
12.3	Logical Operators in Regex	135
12.3.1	Logical OR	135
12.3.2	Logical NOT	136
12.3.3	Logical AND	136
13	Regex Functions in "stringr"	138
13.1	Detecting patterns with <code>str_detect()</code>	138
13.2	Extract first match with <code>str_extract()</code>	139
13.3	Extract all matches with <code>str_extract_all()</code>	140
13.4	Extract first match group with <code>str_match()</code>	140
13.5	Extract all matched groups with <code>str_match_all()</code>	141
13.6	Locate first match with <code>str_locate()</code>	142
13.7	Locate all matches with <code>str_locate_all()</code>	143
13.8	Replace first match with <code>str_replace()</code>	144
13.9	Replace all matches with <code>str_replace_all()</code>	144
13.10	String splitting with <code>str_split()</code>	145
13.11	String splitting with <code>str_split_fixed()</code>	147
14	Regex Functions in R	149
14.1	Pattern Finding Functions	149
14.1.1	Function <code>grep()</code>	149
14.1.2	Function <code>grepl()</code>	151
14.1.3	Function <code>regexpr()</code>	151
14.1.4	Function <code>gregexpr()</code>	152
14.1.5	Function <code>regexec()</code>	153
14.2	Pattern Replacement Functions	155
14.2.1	Replacing first occurrence with <code>sub()</code>	155
14.2.2	Replacing all occurrences with <code>gsub()</code>	156
14.3	Splitting Character Vectors	156

IV	IV Applications	158
15	Fun Plots (part 1)	159
15.1	Me & You Plot	159
16	Fun Plots (part 2)	164
16.1	Colored Jittery Text	164
16.1.1	Assembling the plot	167
17	Basic Examples	169
17.1	Reversing A String	169
17.2	Reversing a String by Characters	170
17.3	Reversing a String by Words	172
17.4	Names of Files	173
17.5	Valid Color Names	174
18	Matching HTML Tags	176
18.1	Attributes <code>href</code>	177
18.1.1	Getting SIG links	178
19	Data Cleaning	181
19.1	Import Data	181
19.2	Extracting Meters	182
19.3	Extracting Country	184
19.4	Cleaning Dates	185
19.5	Month and Day	187
19.6	Extracting Year	188
19.7	Athlete Names	189
20	Data Log File	193
20.1	Reading the text file	194
20.1.1	JPG File Requests	195
20.1.2	Extracting file extensions	196
20.1.3	Image files	197
20.1.4	How to match image files with one regex pattern?	198

About

1st Edition: October 2016 2nd Edition: February 2021 3rd Edition: January 2024



Handling Strings With **R**

Gaston Sanchez

This book aims to provide a panoramic perspective of the wide array of string manipulations that you can perform with R. If you are new to R, or lack experience working with character data, this book will help you get started with the basics of handling strings. Likewise, if you are already familiar with R, you will find material that shows you how to do more advanced string and text processing operations.

About You

I am assuming that you have both R or RStudio installed in your computer. If this is not the case, you can take a look at **Breaking the Ice with R**

<https://www.gastonsanchez.com/R-ice-breaker>

Citation

You can cite this work as:

Sanchez, G. (2024) Handling Strings with R. <https://www.gastonsanchez.com/R-for-strings>

My Series of R Tutorials

This manuscript is part of a series of texts that I've written about Programming and Data Analysis in R:

- **Breaking the Ice with R: Getting Started with R and RStudio** <https://www.gastonsanchez.com/R-ice-breaker>
 - **Tidy Hurricanes: Analyzing Tropical Storms with Tidyverse Tools** <https://www.gastonsanchez.com/R-tidy-hurricanes>
 - **R Coding Basics: An Introduction to the Basics of Coding in R** <https://www.gastonsanchez.com/R-coding-basics>
 - **Rolling Dice: Exploring Simulations in Games of Chance with R** <https://www.gastonsanchez.com/R-rolling-dice>
 - **R for Strings: Handling Strings with R** <https://www.gastonsanchez.com/R-for-strings>
 - **Web Technologies in R: A Short Introduction to Web Technologies in R** <https://www.gastonsanchez.com/R-web-technologies>
-

Donation

As a Data Science and Statistics educator, I love to share the work I do. Each month I spend dozens of hours curating learning materials like this resource. If you find any value and usefulness in it, please consider making a one-time donation—via paypal—in any amount (e.g. the amount you would spend inviting me a cup of coffee or any other drink). Your support really matters.

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Part I

I Characters in R

1 Introductory Appetizer

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

1.1 A Toy Example

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

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

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

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

1.1.1 Abbreviating strings

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

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

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

```
[1] "Alab" "Alas" "Ariz" "Arka" "Cali" "Colo" "Conn" "Dela" "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 five letters we just simply change the argument `minlength = 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
"Okhlm"	"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"

1.1.2 Getting the longest name

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

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

[1] 7 6 7 8 10 8 11 8 7 7 6 5 8 7 4 6 8 9 5 8 13 8 9 11 8
[26] 7 8 6 13 10 10 8 14 12 4 8 6 12 12 14 12 9 5 4 7 8 10 13 9 7

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

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

1.1.3 Selecting States

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

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

```
[1] "Alaska"      "Arkansas"    "Kentucky"    "Nebraska"    "New York"  
[6] "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 involves specifying the argument `ignore.case=TRUE` inside `grep()`:

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

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

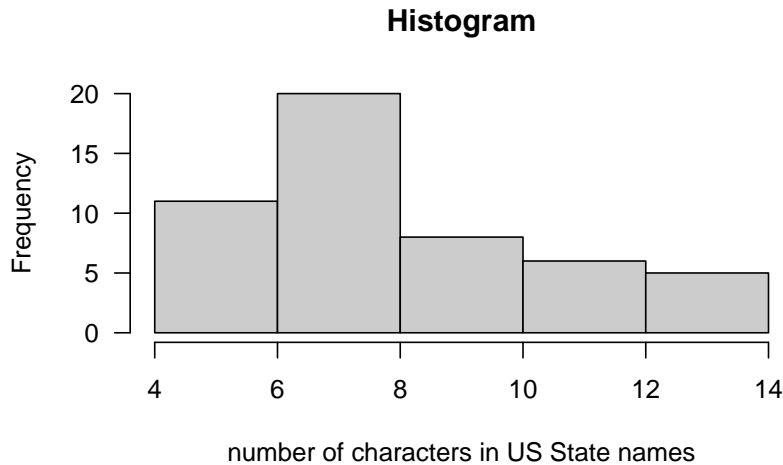
1.1.4 Some computations

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

```
summary(nchar(states))
```

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

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



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

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

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

```
[1] 4 3 2 3 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2 0 2 0 2
[39] 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 2 0 2
[39] 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 2 0 2
[39] 1 2 2 0 1 1 0 1 1 1 0 0
```

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

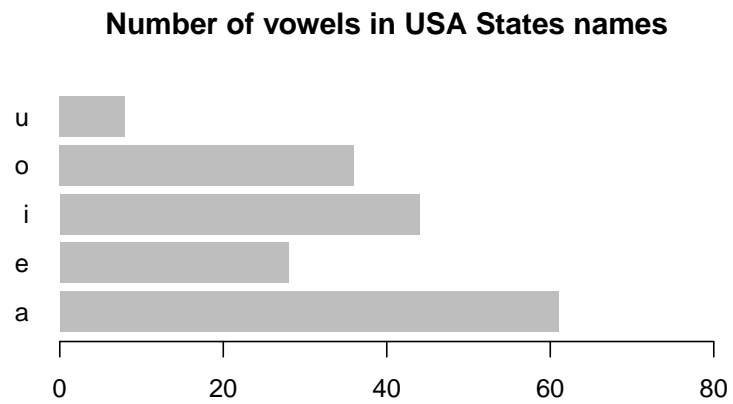
```
# calculate number of vowels in each name
vowels <- c("a", "e", "i", "o", "u")
num_vowels <- vector(mode = "integer", length = 5)

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

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

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

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



2 Character Strings in R

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

2.1 Characters in R

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

You express character strings by surrounding text within double quotes:

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

or you can also surround text within single quotes:

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

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

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

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

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

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

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

```
# single quotes within double quotes
ex1 <- "The 'R' project for statistical computing"

# double quotes within single quotes
ex2 <- 'The "R" project for statistical computing'
```

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

```
ex3 <- "This "is" totally unacceptable"

ex4 <- 'This 'is' absolutely wrong'
```

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

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

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

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

2.2 Getting Started with Strings

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

- names of files and directories
- names of elements in data objects
- text elements displayed in plots and graphs

When you read a file, for instance a data table stored in a csv file, you typically use the `read.table()` function and friends—e.g. `read.csv()`, `read.delim()`. Assuming that the file `dataset.csv` is in your working directory:

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

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

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

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

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

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

2.3 Creating Character Strings

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

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

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

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

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

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

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

```
character(0)
```

```
# check its length
length(example)
```

```
[1] 0
```

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

```
[1] "first"
```

```
# check its length again
length(example)
```

```
[1] 1
```

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

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

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

```
length(example)
```

```
[1] 4
```

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

2.3.1 Empty string

The most basic type of string is 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 <- ""
empty_str
```

```
[1] ""
```

```
# class
class(empty_str)
```

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

2.3.2 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)
empty_chr
```

```
character(0)
```

```
# class
class(empty_chr)
```

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

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

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

```
[1] 1
```

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

```
[1] 0
```

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

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

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

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

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


```
string[0]
```

```
character(0)
```

2.3.3 Function `c()`

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

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

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

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

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

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

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

Related to `character()` R provides two related functions: `as.character()` and `is.character()`. These two functions are methods for coercing objects to type `"character"`, and testing whether an R object is of type `"character"`. For instance, let's define two objects `a` and `b` as follows:

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

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

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

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

```
class(b)
```

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

The function `as.character()` is a coercing method. For better or worse, R allows you to convert (i.e. coerce) non-character objects into character strings with the function `as.character()`:

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

```
[1] "17"
```

2.4 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).

2.4.1 Behavior of R objects with character strings

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

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

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

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

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

```
[1] "1"           "2"           "3"           "4"
[5] "5"           "3.14159265358979" "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 you combine mixed data in vectors, strings will dominate. This means that the mode of the vector will be "character", even if you mix logical values:

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

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

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

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

Matrices. The same behavior of vectors happens when you 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: chr  "a" "b" "c" "d" ...
```

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, you can combine any type of data objects. 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] 0.69608661 -0.52746590 -0.01060646  1.83972471 -0.55306362
```

2.5 The Workhorse Function `paste()`

The function `paste()` is perhaps one of the most important functions that you 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 you 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 you give `paste()` objects of different length, then it will apply a recycling rule. For example, if you paste a single character "X" with the sequence `1:5`, and separator `sep = "."`, this is what you 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
evaluate <- paste("the value of 'e' is", exp(1), NA)

evaluate
```

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

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

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

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

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

2.6 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)

2.6.1 Reading tables

If the data we want to import is in some tabular format (i.e. cells and columns) we can use the set of functions to read tables like `read.table()` and its sister functions, e.g. `read.csv()`, `read.delim()`, `read.fwf()`. These functions read a file in table format and create a data frame from it, with rows corresponding to cases, and columns corresponding to fields in the file.

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

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

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

<http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv>

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

```
# assembling url
abc <- "http://www.abc.net.au/"
radios <- "local/data/public/stations/abc-local-radio.csv"
abc_radios1 <- paste0(abc, radios)

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

In this case, the location of the file is defined in the object `abc` which is the first argument passed to `read.table()`. Then we choose other arguments such as `header = TRUE`, `sep = ","`, and `stringsAsFactors = FALSE`. The argument `header = TRUE` indicates that the first row of the file contains the names of the columns. The separator (a comma) is specified 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 check 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 the structure with `str()` we will get information of each column. The argument `vec.len = 1` indicates that we just want the first element of each variable to be displayed:

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

```
'data.frame':  53 obs. of  18 variables:
 $ State      : chr  "QLD" ...
 $ Website.URL: chr  "http://www.abc.net.au/brisbane/" ...
 $ Station    : chr  "ABC Radio Brisbane" ...
```

```

$ Town           : chr " Brisbane " ...
$ Latitude       : num -27.5 ...
$ Longitude      : num 153 ...
$ Talkback.number : chr "1300 222 612" ...
$ Enquiries.number : chr "07 3377 5222" ...
$ Fax.number     : chr "07 3377 5612" ...
$ Sms.number     : chr "0467 922 612" ...
$ Street.number  : chr "114 Grey Street" ...
$ Street.suburb   : chr "South Brisbane" ...
$ Street.postcode : int 4101 4700 ...
$ PO.box         : chr "GPO Box 9994" ...
$ PO.suburb      : chr "Brisbane" ...
$ PO.postcode    : int 4001 4700 ...
$ Twitter        : chr " abcbrisbane" ...
$ Facebook       : chr " https://www.facebook.com/abcinbrisbane" ...

```

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

2.6.2 Reading raw text

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

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

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

<http://www.textfiles.com/music/ktop100.txt>.

```

# 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 important 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>

3 Basic Manipulations with "base" Functions

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

3.1 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:

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

3.1.1 Count number of characters with `nchar()`

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

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

```
[1]  3  4 11
```

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

```
[1] 20
```

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

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

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

```
[1] 3
```

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

```
[1] 1
```

3.1.2 Convert to lower case with `tolower()`

R comes with three functions for text casefolding.

1. `tolower()`
2. `toupper()`
3. `casefold()`

The first function we'll discuss is `tolower()` which converts any upper case characters into lower case:

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

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

3.1.3 Convert to upper case with toupper()

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

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

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

3.1.4 Upper or lower case conversion with casefold()

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

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

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

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

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

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

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

I've found the case-folding functions to be very helpful when I write functions that take a character input which may be specified in lower or upper case, or perhaps as a mix of both cases. For instance, consider the function `temp_convert()` that takes a temperature value in Fahrenheit degrees, and a character string indicating the name of the scale to be converted.

```
temp_convert <- function(deg = 1, to = "celsius") {
  switch(to,
```

```

    "celsius" = (deg - 32) * (5/9),
    "kelvin" = (deg + 459.67) * (5/9),
    "reaumur" = (deg - 32) * (4/9),
    "rankine" = deg + 459.67)
}

```

Here is how you call `temp_convert()` to convert 10 Fahrenheit degrees into celsius degrees:

```
temp_convert(deg = 10, to = "celsius")
```

```
[1] -12.22222
```

`temp_convert()` works fine when the argument `to = 'celsius'`. But what happens if you try `temp_convert(30, 'Celsius')` or `temp_convert(30, 'CELSIUS')`?

To have a more flexible function `temp_convert()` you can apply `tolower()` to the argument `to`, and in this way guarantee that the provided string by the user is always in lower case:

```

temp_convert <- function(deg = 1, to = "celsius") {
  switch(tolower(to),
    "celsius" = (deg - 32) * (5/9),
    "kelvin" = (deg + 459.67) * (5/9),
    "reaumur" = (deg - 32) * (4/9),
    "rankine" = deg + 459.67)
}

```

Now all the following three calls are equivalent:

```

temp_convert(30, 'celsius')
temp_convert(30, 'Celsius')
temp_convert(30, 'CELSIUS')

```

3.1.5 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 "This is a boring string":

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

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

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

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

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

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

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

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

3.1.6 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
```

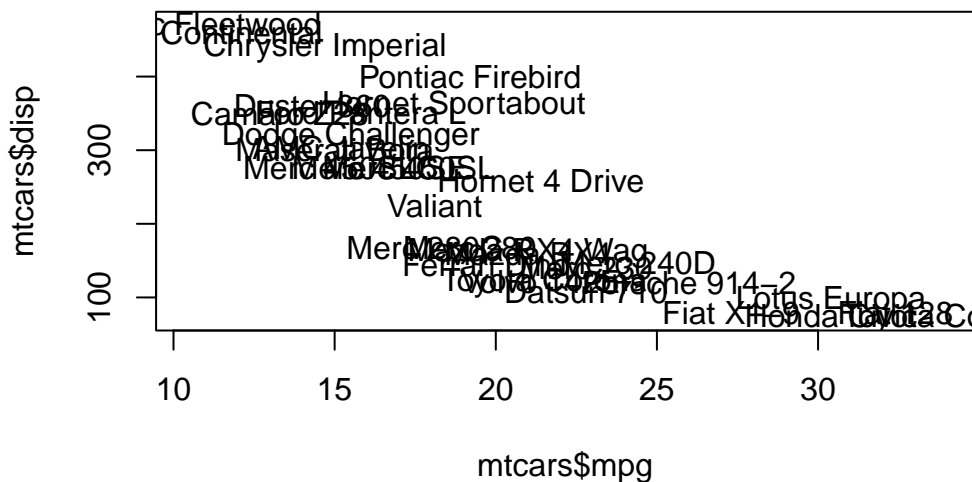
```
white      aliceblue  antiquewhite antiquewhite1
"white"    "alcbl"       "antqw"     "antq1"
```

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

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

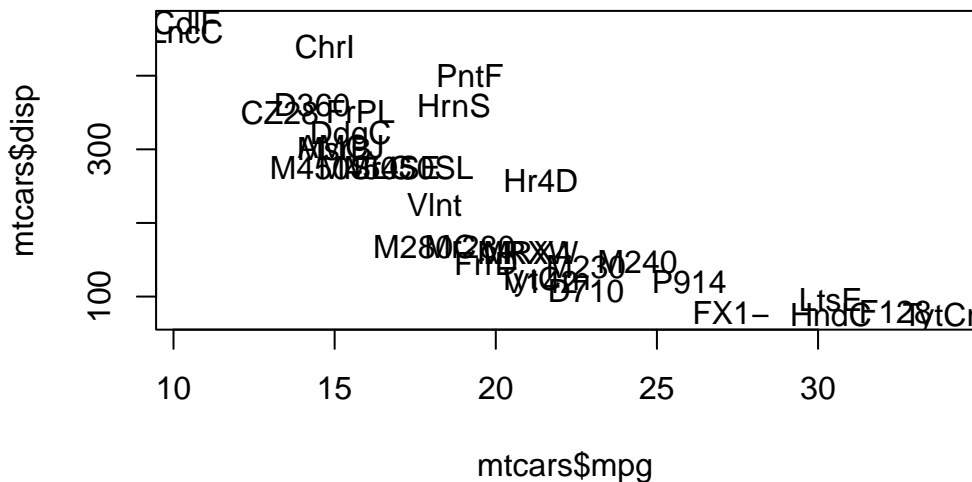
A common use for `abbreviate()` is when plotting names of objects or variables in a graphic. I will use the built-in data set `mtcars` to show you a simple example with a scatterplot between variables `mpg` and `disp`

```
plot(mtcars$mpg, mtcars$disp, type = "n")
text(mtcars$mpg, mtcars$disp, rownames(mtcars))
```



The names of the cars are all over the plot. In this situation you may want to consider using `abbreviate()` to shrink the names of the cars and produce a less “crowded” plot:

```
plot(mtcars$mpg, mtcars$disp, type = "n")
text(mtcars$mpg, mtcars$disp, abbreviate(rownames(mtcars)))
```



3.1.7 Replace substrings with `substr()`

One common operation when working with strings is the extraction and replacement of some characters. There are various ways in which characters can be replaced. If the replacement is based on the positions that characters occupy in the string, you can use the functions `substr()` and `substring()`

`substr()` 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`"     "f#rce" "b`"     "w#th"   "y`"
```

3.1.8 Replace substrings with substring()

Closely related to `substr()` is the function `substring()` which 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
text6 <- c("more", "emotions", "are", "better", "than", "less")
substring(text6, 1:3) <- c(" ", "zzz")
text6
```

```
[1] " ore"      "eazzions" "ar "      "zzzter"   "t an"     "lezz"
```

3.2 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.

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

3.2.1 Set union with `union()`

Let's start our reviewing of set functions with `union()`. As its name indicates, you can use `union()` when you 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.

3.2.2 Set intersection with `intersect()`

Set intersection is performed with the function `intersect()`. You can use this function when you 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"
```

3.2.3 Set difference with `setdiff()`

Related to the intersection, you 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"
```

3.2.4 Set equality with `setequal()`

The function `setequal()` allows you 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
```

3.2.5 Exact equality with `identical()`

Sometimes `setequal()` is not always what we want to use. It might be the case that you 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 you consult the help documentation of `identical()`, you will see that this function is the “safe and reliable way to test two objects for being exactly equal”.

3.2.6 Element contained with `is.element()`

If you wish to test if an element is contained in a given set of character strings you 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, you 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
```

3.2.7 Sorting with `sort()`

The function `sort()` allows you 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 you 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"
```

3.2.8 Repetition with `paste(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"
```

4 Basic Manipulations with "stringr" Functions

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. Sometimes, however, they have an awkward behavior.

As an example, consider the function `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 and company, 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

<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.

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

Function	Description	Similar to
<code>str_c()</code>	string concatenation	<code>paste()</code>

Function	Description	Similar to
<code>str_length()</code>	number of characters	<code>nchar()</code>
<code>str_sub()</code>	extracts substrings	<code>substring()</code>
<code>str_dup()</code>	duplicates characters	<i>none</i>
<code>str_trim()</code>	removes leading and trailing whitespace	<i>none</i>
<code>str_pad()</code>	pads a string	<i>none</i>
<code>str_wrap()</code>	wraps a string paragraph	<code>strwrap()</code>
<code>str_trim()</code>	trims a string	<i>none</i>

Notice that all functions start with "**str_**" followed by a term associated to the task they perform. For example, `str_length()` gives you 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 you 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 "" which is a more common separator when *pasting* strings:

```
# 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))
```

```
character(0)
```

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

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

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

```
[1] "MayTheForceBeWithYou_"
```

```
# synonym function 'str_glue'
str_glue("May", "The", "Force", "Be", "With", "You", .sep = "_")
```

```
May_The_Force_Be_With_You
```

As you can see from the previous examples, an alternative for `str _()` is `str_glue()` with the argument `.sep`.

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 NAs.

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

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

```
[1] 3 3 5 NA 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 <- 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 in nchar(some_factor): '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:

```
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:

```
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 you want to duplicate. The second input, `times`, is the number of times to duplicate each string:

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

```
[1] "holaholahola"
```

```
# use with different '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"    "ametametametamet"
```

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 you want to display the text of `some_quote` within some pre-specified column width (e.g. width of 30). You 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, you may also wish to add some indentation. Here's how you 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, you end up with words that have blank spaces, called *whitespaces*, on either end of the word. In this situation, you 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 trimmed, 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 you 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 the chapters about [regular expressions](#).

Part II

II Print & Format

5 Formatting Text and Numbers

A common task when working with character strings involves printing and displaying them on the screen or on a file. In this chapter you will learn about the different functions and options in R to print strings in a wide variety of common—and not so common—formats.

5.1 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 output either in the R console or to a given file. In this chapter we will describe the following print-related functions:

Function	Description
<code>print()</code>	generic printing
<code>noquote()</code>	print with no quotes
<code>cat()</code>	concatenation
<code>format()</code>	special formats
<code>toString()</code>	convert to string
<code>sprintf()</code>	C-style printing

5.1.1 Generic printing 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(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 you want to print character strings with no quotes you 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.

When to use `print()`

When you type the name of an object in the R console, R calls the corresponding `print` method associated to the class of the object. If the object is a `"data.frame"`, then R will dispatch the method `print.data.frame` and display the output on screen accordingly.

Most of the times you don't really need to invoke `print()`. Usually, simply typing the name of the object will suffice. So when do you actually call `print()`? You use `print()` when your code is inside an **R expression** (i.e. code inside curly braces `{ }`) and you want to see the results of one or more computational steps. Typical examples that require an explicit call to `print()` is when you are interested in looking at some value within a loop, or a conditional structure.

Consider the following dummy `for` loop. It iterates five times, each time adding 1 to the value of the iterator `i`:

```
for (i in 1:5) {
  i + 1
}
```

The above code works and R executes the additions, but nothing is displayed on the console. This is because the command `i + 1` forms part of an R expression, that is, it is within the braces `{ }`. To be able to see the actual computations you should call `print()` like so:

```
for (i in 1:5) {
  print(i + 1)
}
```

```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```

5.1.2 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 name 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
```

5.1.3 Concatenate and print with `cat()`

Another very useful function is `cat()` which allows you 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 R objects (typically vectors). However, when we pass numeric and/or complex vectors, 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 you 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 character vector that will be included to separate the concatenated elements:

```
# specifying '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

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

January-February-March-April

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

JanuaryFebruaryMarchApril

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)
```

Looooooooooooong 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. This is done by specifying the `file` argument:

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

5.1.4 Encoding strings with `format()`

The function `format()` allows you 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] "13.12346"
```

Some useful arguments used in `format()`:

- `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 controlling 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

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

```
[1] "13.700"
```

```
# use of 'digits'
format(c(6.0, 13.1), digits = 2)
```

```
[1] " 6" "13"
```

```
# use of 'digits' and 'nsmall'
format(c(6.0, 13.1), digits = 2, nsmall = 1)
```

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

By default, `format()` pads the strings with spaces so that they all have 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"
```


6 C-style Formatting

R comes with the `sprintf()` function that provides string formatting like in the **C** language. To be more precise, this function is a wrapper for the C library function of the same name. In many other programming languages, this type of printing is known as *printf* which stands for **print formatting**. Simply put, `sprintf()` allows you to create strings as output using formatted data.

The function `sprintf()` requires using a special syntax that may look awkward the first time you use it. Here is one example:

```
sprintf("I woke up at %s:%s%s a.m.", 8, 0, 5)
```

```
[1] "I woke up at 8:05 a.m."
```

How does `sprintf()` work? The first argument of this function is a character vector of one element that contains the text to be formatted. Observe that inside the text there are various percent symbols `%` followed by the letter **s**. Each `%` is referred to as a **slot**, which is basically a placeholder for a variable that will be formatted. The rest of the inputs passed to `sprintf()` are the values that will be used in each of the slots.

The string in the previous example contains three slots of the same type, `%s`, and the subsequent arguments are numbers 8, 0, and 5. Each number is used as a value for each slot. The letter **s** indicates that the formatted variable is specified as a *string*.

Most of the times you won't use `sprintf()` like in the example above. Instead, what you will pass are variables containing different values:

```
hour <- 8
mins1 <- 0
mins2 <- 5

sprintf("I woke up at %s:%s%s a.m.", hour, mins1, mins2)
```

```
[1] "I woke up at 8:05 a.m."
```

6.1 C-style Formatting Options

The string format `%s` is just one of a larger list of available formatting options. The following table shows the most common formatting specifications:

Notation	Description
<code>%s</code>	a string
<code>%d</code>	an integer
<code>%0xd</code>	an integer padded with <code>x</code> leading zeros
<code>%f</code>	decimal notation with six decimals
<code>%.xf</code>	floating point number with <code>x</code> digits after decimal point
<code>%e</code>	compact scientific notation, <code>e</code> in the exponent
<code>%E</code>	compact scientific notation, <code>E</code> in the exponent
<code>%g</code>	compact decimal or scientific notation (with <code>e</code>)

6.1.1 Format Slot Syntax

The full syntax for a format slot is defined by:

`%[parameter][flags][width][.precision][length]type`

The percent symbol, `%`, as we said, indicates a placeholder or slot.

The **parameter** field is an optional field that can take the value `n$` in which `n` is the number of the variable to display, allowing the variables provided to be used multiple times, using varying format specifiers or in different orders.

```
sprintf("The second number is %2$d, the first number is %1$d", 2, 1)
```

```
[1] "The second number is 1, the first number is 2"
```

The **flags** field can be zero or more (in any order) of:

- `-` (minus) Left-align the output of this placeholder.
- `+` (plus) Prepends a plus for positive signed-numeric types.
- `' '` (space) Prepends a space for positive signed-numeric types.
- `0` (zero) When the 'width' option is specified, prepends zeros for numeric types.
- `#` (hash) Alternate form:
 - for `g` and `G` types, trailing zeros are not removed.

- for `f`, `F`, `e`, `E`, `g`, `G` types, the output always contain a decimal point.
- for `o`, `x`, `X` types, the text `0`, `0x`, `0X`, respectively, is prepended to non-zero numbers.

The `width` field is an optional field that you use to specify a minimum number of characters to output, and is typically used to pad fixed-width fields in tabulated output, where the fields would otherwise be smaller, although it does not cause truncation of oversized fields.

```
sprintf("%*d", 5, 10)
```

```
[1] "    10"
```

The `precision` field usually specifies a maximum limit on the output, depending on the particular formatting type.

```
sprintf("%.3s", 3, "abcdef")
```

```
[1] "abc"
```

The `length` field is also optional, and can be any of:

The most important field is the `type` field.

- `%`: Prints a literal `%` character (this type doesn't accept any flags, width, precision, length fields).
- `d`, `i`: integer value as signed decimal number.
- `f`: double value in normal fixed-point notation.
- `e`, `E`: double value in standard form.
- `g`, `G`: double value in either normal or exponential notation.
- `x`, `X`: unsigned integer as a hexadecimal number. `x` uses lower case, while `X` uses upper case.
- `o`: unsigned integer in octal notation.
- `s`: null terminated string.
- `a`, `A`: double value in hexadecimal notation

6.1.2 Example: basic `sprintf()`

Let's begin with a minimal example to explore the different formatting options of `sprintf()`. Consider a real fraction like $1/6$; in R the default output of this fraction will be:

1 / 6

```
[1] 0.1666667
```

Notice that $1/6$ is printed with seven decimal digits. The number $1/6$ is actually an irrational number and so the computer needs to round it to some number of decimal digits. You can modify the default printing format in several ways. One option is to display only six decimal digits with the `%f` option:

```
# print 6 decimals
sprintf('%f', 1/6)
```

```
[1] "0.166667"
```

But you can also specify a different number of decimal digits, say 3. This can be achieved specifying an option of `%.3f`:

```
# print 3 decimals
sprintf('%.3f', 1/6)
```

```
[1] "0.167"
```

The table below shows six different outputs for $1/6$

Notation	Output
<code>%s</code>	0.166666666666667
<code>%f</code>	0.166667
<code>%.3f</code>	0.167
<code>%e</code>	1.666667e-01
<code>%E</code>	1.666667E-01
<code>%g</code>	0.166667

When would you use `sprintf()`? Everytime you produce output text. Some cases include:

- 1) exporting output to some file.
- 2) printing output on console.
- 3) forming new strings.

6.1.3 Example: File Names

When working on data analysis projects, it is common to generate different files with similar names (e.g. either for creating images, or data files, or documents). Imagine that you need to generate the names of 3 data files (with .csv extension). All the files have the same prefix name but each of them has a different number: `data01.csv`, `data02.csv`, and `data03.csv`. One naive solution to generate a character vector with these names in R would be to write something like this:

```
file_names <- c('data01.csv', 'data02.csv', 'data03.csv')
```

Instead of writing each file name, you can generate the vector `file_names` in a more efficient way taking advantage of the vectorized nature of `paste0()`:

```
file_names <- paste0('data0', 1:3, '.csv')

file_names
```

```
[1] "data01.csv" "data02.csv" "data03.csv"
```

Now imagine that you need to generate 100 file names numbered from 01, 02, 03, to 100. You could write a vector with 100 file names but it's going to take you a while. A preferable solution is to use `paste0()` like in the approach of the previous example. In this case however, you would need to create two separate vectors—one with numbers 01 to 09, and another one with numbers 10 to 100—and then concatenate them in one single vector:

```
files1 <- paste0('data0', 1:9, '.csv')
files2 <- paste0('data', 10:100, '.csv')
file_names <- c(files1, files2)
```

Instead of using `paste0()` to create two vectors, you can use `sprintf()` with the `%0xd` option to indicate that an integer should be padded with `x` leading zeros. For instance, the first nine file names can be generated as:

```
sprintf('data%02d.csv', 1:9)
```

```
[1] "data01.csv" "data02.csv" "data03.csv" "data04.csv" "data05.csv"
[6] "data06.csv" "data07.csv" "data08.csv" "data09.csv"
```

To generate the 100 file names do:

```
file_names <- sprintf('data%02d.csv', 1:100)
```

The first nine elements in `file_names` will include a leading zero before the integer; the following elements will not include the leading zero.

6.1.4 Example: Fahrenheit to Celsius

This example involves working on a function to convert Fahrenheit degrees into Celsius degrees. The conversion formula is:

$$Celsius = (Fahrenheit - 32) \times \frac{5}{9}$$

You can define a simple function `to_celsius()` that takes one argument, `temp`, which is a number representing temperature in Fahrenheit degrees. This function will return the temperature in Celsius degrees:

```
to_celsius <- function(temp = 1) {  
  (temp - 32) * 5/9  
}
```

You can use `to_celsius()` as any other function in R. Say you want to know how many Celsius degrees are 95 Fahrenheit degrees:

```
to_celsius(95)
```

```
[1] 35
```

To make things more interesting, let's create another function that not only computes the temperature conversion but also prints a more informative message, something like: 95 Fahrenheit degrees = 35 Celsius degrees.

We'll name this function `fahrenheit2celsius()`:

```
fahrenheit2celsius <- function(temp = 1) {  
  celsius <- to_celsius(temp)  
  sprintf('%.2f Fahrenheit degrees = %.2f Celsius degrees', temp, celsius)  
}
```

Notice that `fahrenheit2celsius()` makes use of `to_celsius()` to compute the Celsius degrees. And then `sprintf()` is used with the options `%.2f` to display the temperatures with two decimal digits. Try it out:

```
fahrenheit2celsius(95)
```

```
[1] "95.00 Fahrenheit degrees = 35.00 Celsius degrees"
```

```
fahrenheit2celsius(50)
```

```
[1] "50.00 Fahrenheit degrees = 10.00 Celsius degrees"
```

6.1.5 Example: Car Traveled Distance

Our third example is a little bit more sophisticated. The idea is to construct an object of class "car" that contains characteristics like the name of the car, its make, its year, and its fuel consumption in *city*, *highway* and *combined*.

Let's consider a *Mazda 3* for this example. One possible way to define a "car" object is to use a list with the following elements:

```
mazda3 <- list(  
  name = 'mazda3', # car name  
  make = 'mazda',  # car make  
  year = 2015,     # year model  
  city = 30,       # fuel consumption in city  
  highway = 40,    # fuel consumption in highway  
  combined = 33)   # fuel consumption combined (city-and-hwy)
```

So far we have an object `mazda3` that is essentially a list. Because we want to create a `print()` method for objects of class "car" we need to assign this class to our `mazda3`:

```
class(mazda3) <- "car"
```

Now that we have our "car" object, we can create a `print.car()` function. In this way, everytime we type `mazda3`, instead of getting the typical list output, we will get a customized display:

```
print.car <- function(x) {
  cat("Car\n")
  cat(sprintf('name: %s\n', x$name))
  cat(sprintf('make: %s\n', x$make))
  cat(sprintf('year: %s\n', x$year))
  invisible(x)
}
```

Next time you type `mazda3` in your console, R will display these lines:

```
mazda3
```

```
Car
name: mazda3
make: mazda
year: 2015
```

It would be nice to have a function `miles()` that allows you to calculate the traveled distance for a given amount of gas (in gallons), taking into account the type of fuel consumption (e.g. city, highway, combined):

```
miles <- function(car, fuel = 1, mpg = 'city') {
  stopifnot(class(car) == 'car')
  switch(mpg,
    'city' = car$city * fuel,
    'highway' = car$highway * fuel,
    'combined' = car$combined * fuel,
    car$city * fuel)
}
```

The `miles()` function takes three parameters: `car` is an object of class "car", `fuel` is the number of gallons, and `mpg` is the type of fuel consumption ('city', 'highway', 'combined'). The first command checks whether the first parameter is an object of class "car". If it is not, then the function will stop the execution raising an error. The second command involves using the function `switch()` to compute the traveled miles. It switches to the corresponding consumption depending on the provided value of `mpg`. Note that the very last switch condition is a *safety* condition in case the user mispecifies `mpg`.

Let's say you want to know how many miles the `mazda3` could travel with 4 gallons of gas depending on the different types of consumption:


```
miles(mazda3, fuel = 4, 'city')
```

```
[1] 120
```

```
miles(mazda3, fuel = 4, 'highway')
```

```
[1] 160
```

```
miles(mazda3, fuel = 4, 'combined')
```

```
[1] 132
```

Again, to make things more user friendly, we are going to create a function `get_distance()` that prints a more informative message about the traveled distance:

```
get_distance <- function(car, fuel = 1, mpg = 'city') {  
  distance <- miles(car, fuel = fuel, mpg = mpg)  
  cat(sprintf('A %s can travel %s miles\n',  
              car$name, distance))  
  cat(sprintf('with %s gallons of gas\n', fuel))  
  cat(sprintf('using %s consumption', mpg))  
}
```

And this is how the output when calling `get_distance` looks like:

```
get_distance(mazda3, 4, 'city')
```

```
A mazda3 can travel 120 miles  
with 4 gallons of gas  
using city consumption
```

6.1.6 Example: Coffee Prices

Consider some coffee drinks and their prices. We'll put this information in a vector like this:

```
prices <- c(
  'americano' = 2,
  'latte' = 2.75,
  'mocha' = 3.45,
  'capuccino' = 3.25)
```

What type of vector is `prices`? Is it a character vector? Is it numeric vector? Or is it some sort of vector with mix-data? We have seen that vectors are *atomic* structures, meaning that all their elements must be of the same class. So `prices` is definitely not a vector with mix-data. From the code chunk we can observe that each element of the vector is formed by a string, followed by the `=` sign, followed by some number. This way of defining a vector is not very common in R but it is perfectly valid. Each string represents the name of an element, while the numbers are the actual elements. Therefore `prices` is in reality a numeric vector. You can confirm this by looking at the mode (or data type):

```
mode(prices)
```

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

Let's say you want to list the names of the coffees and their prices. If you just simply try to `print()` the prices, the output will be the entire vector `prices`:

```
print(prices)
```

```
americano    latte    mocha capuccino
      2.00      2.75      3.45      3.25
```

Alternatively, you can use a for loop to `print()` each individual element of the vector `prices`, but again the output is displayed in an awkward fashion:

```
for (p in seq_along(prices)) {
  print(prices[p])
}
```

```
americano
      2
latte
      2.75
```

```
mocha
  3.45
capuccino
  3.25
```

To list the names of the coffees and their prices, it would be nicer to use a combination of `paste0()` and `print()`. In addition, you can be more descriptive adding some auxiliary text such that the output prints something like: *“americano has a price of \$2”*.

```
for (p in seq_along(prices)) {
  print(paste0(names(prices)[p], ' has price of $', prices[p]))
}
```

```
[1] "americano has price of $2"
[1] "latte has price of $2.75"
[1] "mocha has price of $3.45"
[1] "capuccino has price of $3.25"
```

Another possible solution consists of combining `print()` and `sprintf()`:

```
for (p in seq_along(prices)) {
  print(sprintf('%s has price of $%s', names(prices)[p], prices[p]))
}
```

```
[1] "americano has price of $2"
[1] "latte has price of $2.75"
[1] "mocha has price of $3.45"
[1] "capuccino has price of $3.25"
```

One limitation of `quote()` is that it won't work inside a for loop:

```
for (p in seq_along(prices)) {
  noquote(sprintf('%s has price of $%s', names(prices)[p], prices[p]))
}
```

If what you want is to print the output without quotes, then you need to use `cat()`; just make sure to add a newline character `"\n"`:

```
for (p in seq_along(prices)) {
  cat(sprintf('%s has price of $%s\n', names(prices)[p], prices[p]))
}
```

```
americano has price of $2
latte has price of $2.75
mocha has price of $3.45
capuccino has price of $3.25
```

6.1.7 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 objects 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...."
```

6.1.8 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
```

7 Input and Output

7.1 Output

Some times you need to export results to a file. Typically this happens when you want to export a data table to a text file. R provides the functions `write.table()` and `write.csv()` for these purposes. These functions let you send a matrix or a data frame to a text file that will have a tabular format (i.e. rows and columns).

7.1.1 Concatenating output

You can use `cat()` to concatenate and print information to a file.

To show you how to use `cat()` let's illustrate a simple example using the data frame `mtcars` that comes in R.

```
# summary statistics of unemp
min(mtcars$mpg)
max(mtcars$mpg)
median(mtcars$mpg)
mean(mtcars$mpg)
sd(mtcars$mpg)
```

The goal is to generate a file `mpg-statistics.txt` with the following contents:

Miles per Gallon Statistics

```
Minimum: 10.40
Maximum: 33.90
Median  : 19.20
Mean    : 20.09
Std Dev: 6.02
```

Here is one way to do it. First, let's assign the statistics to different objects:

```
# summary statistics of mpg
mpg_min <- min(mtcars$mpg)
mpg_max <- max(mtcars$mpg)
mpg_med <- median(mtcars$mpg)
mpg_avg <- mean(mtcars$mpg)
mpg_sd <- sd(mtcars$mpg)
```

After creating the objects containing the summary statistics, the next step is to export them to the text file `mpg-statistics.txt` via `cat()`. Assuming that the output file is in your working directory, here's how you can send the set of strings to the text file:

```
# name of output file
outfile <- "mpg-statistics.txt"

# first line of the file
cat("Miles per Gallon Statistics\n\n", file = outfile)

# subsequent lines appended to the output file
cat("Minimum:", mpg_min, "\n", file = outfile, append = TRUE)
cat("Maximum:", mpg_max, "\n", file = outfile, append = TRUE)
cat("Median :", mpg_med, "\n", file = outfile, append = TRUE)
cat("Mean   :", mpg_avg, "\n", file = outfile, append = TRUE)
cat("Std Dev:", mpg_sd, "\n", file = outfile, append = TRUE)
```

The first line exported to `mpg-statistics.txt` is a string with the title "Miles per Gallon Statistics\n\n". Observe that we are using two new line characters "\n\n" to add some space between the title and the statistics. The rest of calls to `cat()` use the argument `append = TRUE` to concatenate the specified strings to the end of the text file without overriding the existing lines.

If you run the code of this example and look at the contents of `mpg-statistics.txt`, you will see the following output:

Miles per Gallon Statistics

```
Minimum: 10.4
Maximum: 33.9
Median  : 19.2
Mean    : 20.09062
Std Dev: 6.026948
```


As you can tell, the displayed values have a different number of decimal digits. If you just want to keep two decimal digits, you can use `sprintf()` and choose the format `"%0.2f"`. Let's re-export the lines:

```
cat("Miles per Gallon Statistics\n\n", file = outfile)
cat(sprintf('Minimum: %0.2f', mpg_min), "\n", file = outfile, append = TRUE)
cat(sprintf('Maximum: %0.2f', mpg_max), "\n", file = outfile, append = TRUE)
cat(sprintf('Median : %0.2f', mpg_med), "\n", file = outfile, append = TRUE)
cat(sprintf('Mean   : %0.2f', mpg_avg), "\n", file = outfile, append = TRUE)
cat(sprintf('Std Dev: %0.2f', mpg_sd), "\n", file = outfile, append = TRUE)
```

Now the content of `mpg-statistics.txt` should look like this:

Miles per Gallon Statistics

Minimum: 10.40
Maximum: 33.90
Median : 19.20
Mean : 20.09
Std Dev: 6.03

Here is an exercise for you: How would you avoid writing that many calls to `cat()`?

7.1.2 Sinking output

Another interesting function is `sink()`. This function is very useful when you want to export R output as is displayed in the R console. For example, consider the output from `summary()`

```
summary(mtcars)
```

You could assign the output of `summary(mtcars)` to an object and then try `writeLines()` to export the results to a file `mtcars-summary.txt`, but you won't keep the same format of R:

```
mtcars_summary <- summary(mtcars)
writeLines(mtcars_summary, con = "mtcars-summary.txt")
```

To be able to keep the same output display of R, you must use `sink()`. This function will **divert** R output to the specified file:

```
sink(file = "mtcars-statistics.txt")
summary(mtcars)
sink()
```

Your turn: Use `sink()` to send the output from running a linear regression of `mpg` on `disp` with the function `lm()`. Also export the results from using `summary()` on the regression object. And/or try running a t-test between `mpg` and `disp` with `t.test()`.

7.2 Exporting Tables

Another interesting tool to export tables in LaTeX or HTML formats is provided by the R package "xtable" and its main function `xtable()`.

```
library(xtable)

# linear regression
reg <- lm(mpg ~ disp, data = mtcars)

# create xtable and export it
reg_table <- xtable(reg)
```

The object `reg_table` is an object of class "xtable". What you do with this type of objects is `print()` them to a file.

To print `reg_table` in latex format to a `.tex` file:

```
print(reg_table, type = "latex", file = "reg-table.tex")
```

To print `reg_table` in html format to an `.html` file:

```
print(reg_table, type = "html", file = "reg-table.html")
```

Part III

III Regex

8 Getting Started with Regular Expressions

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

In this chapter, we use functions from the package "stringr"

```
library(stringr)
```

8.1 What are Regular Expressions?

The name “Regular Expression” does not say much. However, regular expressions are all about text. Think about how much text is all around you in our modern digital world: email, text messages, news articles, blogs, computer code, contacts in your address book—all these things are text. Regular expressions are a tool that allows us to work with these text by describing text patterns.

A **regular expression** is a special text string for describing a certain amount of text. This “certain amount of text” receives the formal name of *pattern*. In other words, a regular expression is a set of symbols that describes a text pattern. More formally we say that a regular expression is a *pattern that describes a set of strings*. In addition to this first meaning, the term regular expression can also be used in a slightly different but related way: as the formal language of these symbols that needs to be interpreted by a regular expression processor. Because the term “regular expression” is rather long, most people use the word **regex** as a shortcut term. And you will even find the plural *regexes*.

It is also worth noting what regular expressions are not. They’re not a programming language. They may look like some sort of programming language because they are a formal language with a defined set of rules that gets a computer to do what we want it to do. However, there are no variables in regex and you can’t do computations like adding $1 + 1$.

8.1.1 What are Regular Expressions used for?

We use regular expressions to work with text. Some of its common uses involve testing if a phone number has the correct number of digits, if a date follows a specific format (e.g. mm/dd/yy), if an email address is in a valid format, or if a password has numbers and special characters. You could also use regular expressions to search a document for *gray* spelt either as “gray” or “grey”. You could search a document and replace all occurrences of “Will”, “Bill”, or “W.” with William. Or you could count the number of times in a document that the word “analysis” is immediately preceded by the words “data”, “computer” or “statistical” only in those cases. You could use it to convert a comma-delimited file into a tab-delimited file or to find duplicate words in a text.

In each of these cases, you are going to use a regular expression to write up a description of what you are looking for using symbols. In the case of a phone number, that pattern might be three digits followed by a dash, followed by three digits and another dash, followed by four digits. Once you have defined a pattern then the regex processor will use our description to return matching results, or in the case of the test, to return true or false for whether or not it matched.

8.1.2 A word of caution about regex

If you have never used regular expressions before, their syntax may seem a bit scary and cryptic. You will see strings formed by a bunch of letters, digits, and other punctuation symbols combined in seemingly nonsensical ways. As with any other topic that has to do with programming and data analysis, learning the principles of regex and becoming fluent in defining regex patterns takes time and requires a lot of practice. The more you use them, the better you will become at defining more complex patterns and getting the most out of them.

Regular Expressions is a wide topic and there are books entirely dedicated to this subject. The material offered in this book is not extensive and there are many subtopics that I don’t cover here. Despite the initial barriers that you may encounter when entering the regex world, the pain and frustration of learning this tool will payoff in your data science career.

8.1.3 About Regular Expressions in R

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 but it does not provide the wide range of capabilities that other scripting languages do. Nevertheless, they can take you very far with some workarounds (and a bit of patience).

Although I am assuming that you are new to regex, I won’t cover everything there is to know about regular expressions. Instead, I will focus on how R works with regular expressions, as well as the R syntax that you will have to use for regex operations.

One of the best tools you must have in your toolkit is the R package "**stringr**" (by Hadley Wickham). It provides functions that have similar behavior to those of the base distribution in R. But it also provides many more facilities for working with regular expressions.

To know more about regular expressions in general, you can find some useful information in the following resources:

- Regex wikipedia: For those readers who have no experience with regular expressions, a good place to start is by checking the wikipedia entrance.

http://en.wikipedia.org/wiki/Regular_expression

- Regular-Expressions.info website (by Jan Goyvaerts): An excellent 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.

<http://www.regular-expressions.info>

- Mastering Regular Expressions (by Jeffrey Friedl): 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 addressed for readers already experienced with regex.

<http://regex.info/book.html>

8.2 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.

8.3 Literal Characters

We're going to start with the simplest match of all: a **literal character**. A literal character match is one in which a given character such as the letter "R" matches the letter *R*. This type of match is the most basic type of regular expression operation: just matching plain text.

8.3.1 Matching Literal Characters

The following examples are extremely basic but they will help you get a good understanding of regex.

Consider the following text stored in a character vector `this_book`:

```
this_book <- 'This book is mine'
```

The first regular expression we are going to work with is "book". This pattern is formed by a letter *b*, followed by a letter *o*, followed by another letter *o*, followed by a letter *k*. As you may guess, this pattern matches the word *book* in the character vector `this_book`. To have a visual representation of the actual pattern that is matched, you should use the function `str_view()` from the package "stringr" (you may need to upgrade to a recent version of RStudio):

```
str_view(this_book, 'book')
```

As you can tell, the pattern "book" doesn't match the entire content in the vector `this_book`; it just matches those four letters.

It may seem really simple but there are a couple of details to be highlighted. The first is that regex searches are case sensitive by default. This means that the pattern "Book" would not match *book* in `this_book`.

```
str_view("This Book is mine.", 'book')
```

You can change the matching task so that it is case insensitive but we will talk about it later.

Let's add more text to `this_book`:

```
this_book <- 'This book is mine. I wrote this book with bookdown.'
```

Let's use `str_view()` to see what pieces of text are matched in `this_book` with the pattern "book":

```
str_view(this_book, "book")
```

As you can tell, only the first occurrence of *book* was matched. This is a common behavior of regular expressions in which they return a match as fast possible. You can think of this behavior as the “eager principle”, that is, regular expressions are eager and they will give preference to an early match. This is a minor but important detail and we will come back to this behavior of regular expressions.

All the letters and digits in the English alphabet are considered literal characters. They are called *literal* because they match themselves.

```
str_view <- c("I had 3 quesadillas for lunch", "3")
```

Here is another example:

```
transport <- c("car", "bike", "boat", "airplane")
```

The first pattern to test is the letter "a":

```
str_view(transport, "a")
```

When you execute the previous command, you should be able to see that the letter "a" is highlighted in the words *car*, *boat* and *airplane*.

8.4 R Functions for Regular Expressions

In order to move on with the discussion of regular expressions, we need to talk about some of the functions available in R for regex.

8.4.1 Regex Functions in "base" Package

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:

Function	Purpose	Characteristic
<code>grep()</code>	finding regex matches	which elements are matched (index or value)
<code>grepl()</code>	finding regex matches	which elements are matched (TRUE,FALSE)
<code>regexpr()</code>	finding regex matches	positions of the first match
<code>gregexpr()</code>	finding regex matches	positions of all matches
<code>regexec()</code>	finding regex matches	hybrid of <code>regexpr()</code> and <code>gregexpr()</code>

Function	Purpose	Characteristic
<code>sub()</code>	replacing regex matches	only first match is replaced
<code>gsub()</code>	replacing regex matches	all matches are replaced
<code>strsplit()</code>	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)
```

8.4.2 Regex Functions in Package "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.

Function	Description
<code>str_detect()</code>	Detect the presence or absence of a pattern in a string
<code>str_extract()</code>	Extract first piece of a string that matches a pattern
<code>str_extract_all()</code>	Extract all pieces of a string that match a pattern
<code>str_match()</code>	Extract first matched group from a string
<code>str_match_all()</code>	Extract all matched groups from a string
<code>str_locate()</code>	Locate the position of the first occurrence of a pattern in a string

Function	Description
<code>str_locate_all()</code>	Locate the position of all occurrences of a pattern in a string
<code>str_replace()</code>	Replace first occurrence of a matched pattern in a string
<code>str_replace_all()</code>	Replace all occurrences of a matched pattern in a string
<code>str_split()</code>	Split up a string into a variable number of pieces
<code>str_split_fixed()</code>	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 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 occurrence, we should use `str_locate()`; to locate the positions of all matches we should use `str_locate_all()`.

8.4.3 Matching Literal Characters With "stringr" Functions

Having introduced the regex functions available in "stringr", let's continue describing how to match literal characters. We had defined a string `this_book`

```
this_book <- 'This book is mine. I wrote this book with bookdown.'
```

We can use the function `str_detect()` to look for the pattern "book"

```
str_detect(string = this_book, pattern = "book")
```

```
[1] TRUE
```

If there is a match, then `{str_detect() }` returns `TRUE`. Conversely, if there is no match, `str_detect()` will return `FALSE`

```
str_detect(string = this_book, pattern = 'tablet')
```

```
[1] FALSE
```

All the letters and digits in the English alphabet are considered literal characters. They are called *literal* because they match themselves.

```
str_detect <- c(string = "I had 3 quesadillas for lunch", pattern = "3")
```

Here is another example:

```
transport <- c("car", "bike", "boat", "airplane")
```

The first pattern to test is the letter "a":

```
str_view(string = transport, pattern = "a")
```

When you execute the previous command, you should be able to see that the letter "a" is highlighted in the words *car*, *boat* and *airplane*.

8.5 Metacharacters

The next topic that you should learn about regular expressions has to do with **metacharacters**. As you just learned, the most basic type of regular expressions are the literal characters which are characters that match themselves. However, not all characters match themselves. Any character that is not a literal character is a metacharacter.

8.5.1 About Metacharacters

Metacharacter are characters that have a special meaning and they allow you to transform literal characters in very interesting ways. Sometimes they act like mathematical operators: transforming literal characters into powerful expressions.

Below is the list of metacharacters in *Extended Regular Expressions* (EREs):

. \ | () [] { } \$ - ^ * + ?

- the dot .
- the backslash \
- the bar |
- left or opening parenthesis (
- right or closing parenthesis)

- left or opening bracket [
- right or closing bracket]
- left or opening brace {
- right or closing brace }
- the dollar sign \$
- the dash, hyphen or minus sign -
- the caret or hat ^
- the star or asterisk *
- the plus sign +
- the question mark ?

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.

We're going to be working with these characters throughout the rest of the book. Simply put, everything else that you need to know about regular expressions besides literal characters is how these metacharacters work. The good news is that there are only a few metacharacters to learn. The bad news is that some metacharacters can have more than one meaning. And learning those meanings definitely takes time and requires hours of practice. The meaning of the metacharacters greatly depend on the context in which you use them, how you use them, and where you use them. If it wasn't enough complication, it is also the metacharacters that have variation between the different regex engines.

8.6 The Wildcard Metacharacter

The first metacharacter you should learn about is the dot or period `"."`, better known as the **wildcard** metacharacter.

Like in many card games where one of the cards in the deck is wild and can be used to replace other types of cards, there is also a wild character in regex that has the same purpose—hence its name.

This metacharacter is used to match **ANY** character except for a new line.

For example, consider the pattern `"p.n"`, that is, *p wildcard n*. This pattern will match *pan*, *pen*, and *pin*, but it will not match *prun* or *plan*. The dot only matches one single character.

```
pns <- c('pan', 'pen', 'pin', 'plan', 'prun', 'p n', 'p\nn')
str_detect(string = pns, pattern = 'p.n')
```

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

Observe that "p.n" even matches the blank space, and the new line character "\n". The reason why it does not match `plan` is because the third character is an `a` and not an `n`.

Let's see another example using the vector `c("not", "note", "knot", "nut")` and the pattern "n.t"

```
not <- c("not", "note", "knot", "nut")
str_view(not, "n.t")
```

the pattern "n.t" matches *not* in the first three elements, and *nut* in the last element.

If you specify a pattern "no.", then just the first three elements in `not` will be matched.

```
str_view(not, "no.")
```

And if you define a pattern "kn.", then only the third element is matched.

```
str_view(not, "kn.")
```

The wild metacharacter is probably the most used metacharacter, and it is also the most abused one, being the source of many mistakes. Here is a basic example with the regular expression formed by "5.00". If you think that this pattern will match five with two decimal places after it, you will be surprised to find out that it not only matches *5.00* but also *5100* and *5-00*. Why? Because "." is the metacharacter that matches absolutely anything. You will learn how to fix this mistake in the next section, but it illustrates an important fact about regular expressions: the challenge consists of matching what you want, but also in matching only what you want. You don't want to specify a pattern that is overly permissive. You want to find the thing you're looking for, but only that thing.

As an experiment, try writing a pattern that will match *silver*, *sliver*, and *slider*. See if you can use wildcards to come up with a regular expression that will match all three of those.

```
sil <- c('silver', 'sliver', 'slider')

# your pattern
pat <- ...

# test it
str_detect(string = sil, pattern = pat)
```

8.6.1 Escaping metacharacters

What if you just want to match the character dot? For example, say you have the following vector:

```
fives <- c("5.00", "5100", "5-00", "5 00")
```

If you try the pattern "5.00", it will match all of the elements in `fives`.

```
str_view(fives, "5.00")
```

To actually match the dot character, what you need to do is **escape** the metacharacter. In most languages, the way to escape a metacharacter is by adding a backslash character in front of the metacharacter: `"\."`. When you use a backslash in front of a metacharacter you are “escaping” the character, this means that the character no longer has a special meaning, and it will match itself.

However, R is a bit different. Instead of using a backslash you have to use two backslashes: `"5\\.00"`. This is because the backslash `"\"`, which is another metacharacter, has a special meaning in R. Therefore, to match just the element `5.00` in `fives` in R, you do it like so:

```
str_view(fives, "5\\.00")
```

The following list shows the general regex 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	"\\ "
\\	the backslash	"\\\\"
^	the caret or hat	"\\^"
[the opening bracket	"\\["
]	the closing bracket	"\\]"
{	the opening brace	"\\{"
}	the closing brace	"\\}"
(the opening parenthesis	"\\("
)	the closing parenthesis	"\\)"

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

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

```
[1] "Peace-Love"
```

```
# dot
str_replace("Peace.Love", "\\.", "")
```

```
[1] "PeaceLove"
```

```
# plus
str_replace("Peace+Love", "\\+", "")
```

```
[1] "PeaceLove"
```

```
# caret
str_replace("Peace^Love", "\\^", "")
```

```
[1] "PeaceLove"
```

```
# vertical bar
str_replace("Peace|Love", "\\|", "")
```

```
[1] "PeaceLove"
```

```
# opening round bracket
str_replace("Peace(Love)", "\\(", "")
```

```
[1] "PeaceLove)"
```

```
# closing round bracket
str_replace("Peace(Love)", "\\)", "")
```

```
[1] "Peace(Love"
```

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

```
[1] "PeaceLove]"
```

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

```
[1] "Peace[Love"
```

```
# opening curly bracket
str_replace("Peace{Love}", "\\{", "")
```

```
[1] "PeaceLove}"
```

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

```
[1] "Peace{Love"
```

```
# double backslash
str_replace("Peace\\Love", "\\\\\\", "")
```

```
[1] "PeaceLove"
```


9 Character Sets

In this chapter we will talk about character sets. You will learn about a couple of more metacharacters, the opening and closing brackets `[]`, that will help you define a character set.

These square brackets indicate a character set which will match any one of the various characters that are inside the set. Keep in mind that a character set will match only one character. The order of the characters inside the set does not matter; what matter is just the presence of the characters inside the brackets. So for example if you have a set defined by `"[AEIOU]"`, that will match any one upper case vowel.

9.1 Defining character sets

Let's bring back the vector `pns`

```
pns <- c('pan', 'pen', 'pin', 'pon', 'pun')
pns
```

```
[1] "pan" "pen" "pin" "pon" "pun"
```

Consider the following pattern that includes a character set: `"p[aeiou]n"`. As you can tell, this pattern is formed by the character `p`, folled by the set `[aeiou]`, followed by the character `n`. What does this pattern match? Let's find out with `str_view()`

```
str_view(pns, "p[aeiou]n")
```

The set `"p[aeiou]n"` matches all elements in `pns`.

For comparison purposes, let's create another vector `pnx`

```
pnx <- c('pan', 'pen', 'pin', 'p0n', 'p.n', 'p1n', 'paun')
pnx
```

```
[1] "pan"  "pen"  "pin"  "p0n"  "p.n"  "p1n"  "paun"
```

And let's test again the pattern `"p[aeiou]n"` to see what elements are matched

```
str_view(pnx, "p[aeiou]n")
```

This time only the first three elements in `pnx` are matched. Notice also that *paun* is not matched. This is because the character set matches only one character, either *a* or *u* but not *au*.

If you are interested in matching all capital letters in English, you can define a set formed as:

```
[ABCDEFGHIJKLMNOPQRSTUVWXYZ]
```

Likewise, you can define a set with only lower case letters in English:

```
[abcdefghijklmnopqrstuvwxyz]
```

If you are interested in matching any digit, you can also specify a character set like this:

```
[0123456789]
```

9.2 Character ranges

The previous examples that show character sets containing all the capital letters or all lower case letters are very convenient but require a lot of typing. **Character ranges** are going to help you solve that problem, by giving you a convenient shortcut based on the dash metacharacter `"-"` to indicate a range of characters. A character range consists of a character set with two characters separated by a dash or minus `"-"` sign.

Let's see how you can reexpress the examples in the previous section as character ranges. The set of all digits can be expressed as a character range using the following pattern:

```
[0-9]
```

Likewise, the set of all lower case letters *abcd...xyz* is compactly represented with the character range:

[a-z]

And the character set of all upper case letters *ABD...XYZ* is formed by

[A-Z]

Note that the dash is only a metacharacter when it is inside a character set; outside the character set it is just a literal dash.

So how do you use character range? To illustrate the concept of character ranges let's create a **basic** vector with some simple strings, and see what the different ranges match:

```
basic <- c('1', 'a', 'A', '&', '-', '^')
```

First, we use the range [0-9] to match any single digit

```
# digits
str_view(basic, '[0-9]')
```

Then we use the range of lower case letters [a-z]

```
# lower case letters
str_view(basic, '[a-z]')
```

And finally the range of upper case letters [A-Z]

```
# upper case letters
str_view(basic, '[A-Z]')
```

Now consider the following vector **triplets**:

```
triplets <- c('123', 'abc', 'ABC', ':-')
```

You can use a series of character ranges to match various occurrences of a certain type of character. For example, to match three consecutive digits you can define a pattern "[0-9][0-9][0-9]"; to match three consecutive lower case letters you can use the pattern "[a-z][a-z][a-z]"; and the same idea applies to a pattern that matches three consecutive upper case letters "[A-Z][A-Z][A-Z]".

```
str_view(triplets, '[0-9][0-9][0-9]')
```

```
str_view(triplets, '[A-Z][A-Z][A-Z]')
```

Observe that the element ":-)" is not matched by any of the character ranges that we have seen so far.

Character ranges can be defined in multiple ways. For example, the range "[1-3]" indicates any one digit 1, 2, or 3. Another range may be defined as "[5-9]" comprising any one digit 5, 6, 7, 8 or 9. The same idea applies to letters. You can define shorter ranges other than "[a-z]". One example is "[a-d]" which consists of any one letter *a*, *b*, *c*, and *d*.

Pattern	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

9.3 Negative Character Sets

A common situation when working with regular expressions consists of matching characters that are NOT part of a certain set. This type of matching can be done using a negative character set: by matching any one character that is not in the set. To define this type of sets you are going to use the metacharacter caret "^". If you are using a QWERTY keyboard, the caret symbol should be located over the key with the number 6.

The caret "^" is one of those metacharacters that have more than one meaning depending on where it appears in a pattern. If you use a caret in the first position inside a character set, e.g. [^aeiou], it means *negation*. In other words, the caret in [^aeiou] means “not any one of lower case vowels.”

Let's use the `basic` vector previously defined:

```
basic <- c('1', 'a', 'A', '&', '-', '^')
basic
```

```
[1] "1" "a" "A" "&" "-" "^"
```

To match those elements that are NOT upper case letters, you define a negative character range "[^A-Z]":

```
str_view(basic, '[^A-Z]')
```

It is important that the caret is the first character inside the character set, otherwise the set is not a negative one:

```
str_view(basic, '[A-Z^]')
```

In the example above, the pattern "[A-Z^]" means "any one upper case letter or the caret character." Which is completely different from the negative set "[^A-Z]" that negates any one upper case letter.

If you want to match any character except the caret, then you need to use a character set with two carets: "[^^]". The first caret works as a negative operator, the second caret is the caret character itself:

```
str_view(basic, '[^^]')
```

9.4 Metacharacters Inside Character Sets

Now that you know what character sets are, how to define character ranges, and how to specify negative character sets, we need to talk about what happens when including metacharacters inside character sets.

Except for the caret in the first position of the character set, any other metacharacter inside a character set is already escaped. This implies that you do not need to escape them using backslashes.

To illustrate the use of metacharacters inside character sets, let's use the `pnx` vector:

```
pnx <- c('pan', 'pen', 'pin', 'p0n', 'p.n', 'p1n', 'paun')
pnx
```

The character set formed by "`p[ae.io]n`" includes the dot character. Remember that, in general, the period is the wildcard metacharacter and it matches any type of character. However, the period in this example is inside a character set, and because of that, it loses its wildcard behavior.

```
str_view(pnx, "p[ae.io]n")
```

As you can tell, `"p[ae.io]n"` matches *pan*, *pen*, *pin* and *p.n*, but not *p0n* or *p1n* because the dot is the literal dot, not a wildcard character anymore.

Not all metacharacters become literal characters when they appear inside a character set. The exceptions are the closing bracket `]`, the dash `-`, the caret `^`, and the backslash `\`.

The closing bracket `]` is used to enclose the character set. Thus, if you want to use a literal right bracket inside a character set you must escape it:

```
"[aei\\[ou]"
```

Remember that in R you use double backslash for escaping purposes. This is also why the backslash `\`, or double backslash in R, does not become a literal character.

Another interesting case has to do with the dash or hyphen `-` character. As you know, the dash inside a character set is used to define a range of characters: e.g. `[0-9]`, `[x-z]`, and `[K-P]`. As a general rule, if you want to include a literal dash as part of a range, you should escape it: `"[a-z\\-]"`.

Let's modify the `basic` vector by adding an opening and ending brackets:

```
basic <- c('1', 'a', 'A', '&', '-', '^', '[', ']')
```

How do you match each of the characters that have a special meaning inside a character set?

```
# matching a literal caret
str_view(basic, "[a\\^]")
```

```
# matching a literal dash
str_view(basic, "[a\\-]")
```

```
# matching a literal opening bracket
str_view(basic, "[a\\[")
```

```
# matching a literal closing bracket
str_view(basic, "[a\\]")
```

9.5 Character Classes

Closely related with character sets and character ranges, regular expressions provide another useful construct called **character classes** which, as their name indicates, are used to match a certain class of characters. The most common character classes in most regex engines are:

Character	Matches	Same as
<code>\\d</code>	any digit	<code>[0-9]</code>
<code>\\D</code>	any nondigit	<code>[^0-9]</code>
<code>\\s</code>	any whitespace character	<code>[\f\n\r\t\v]</code>
<code>\\S</code>	any nonwhitespace character	<code>[^\f\n\r\t\v]</code>
<code>\\w</code>	any character considered part of a word	<code>[a-zA-Z0-9_]</code>
<code>\\W</code>	any character not considered part of a word	<code>[^a-zA-Z0-9_]</code>
<code>\\b</code>	any word boundary	
<code>\\B</code>	any non-(word boundary)	

You can think of character classes as another type of metacharacters, or as shortcuts for special character sets.

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

```
[1] "the dandelion war _010"
```

```
str_replace_all("the dandelion war 2010", "\\d", "_")
```

```
[1] "the dandelion war ____"
```

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

```
[1] "_he dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\D", "_")
```

```
[1] "_____2010"
```

Spaces and non-spaces

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

```
[1] "the_dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\s", "_")
```

```
[1] "the_dandelion_war_2010"
```

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

```
[1] "_he dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\S", "_")
```

```
[1] "___ _ _ _ _"
```

Words and non-words

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

```
[1] "_the dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\b", "_")
```

```
[1] "_the_ _dandelion_ _war_ _2010_"
```



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

```
[1] "t_he dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\B", "_")
```

```
[1] "t_h_e d_a_n_d_e_l_i_o_n w_a_r 2_0_1_0"
```

Word boundaries and non-word-boundaries

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

```
[1] "_he dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\w", "_")
```

```
[1] "___ _ _ _ _"
```

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

```
[1] "the_dandelion war 2010"
```

```
str_replace_all("the dandelion war 2010", "\\W", "_")
```

```
[1] "the_dandelion_war_2010"
```

The following table shows the characters that represent whitespaces:

Character	Description
<code>\f</code>	form feed
<code>\n</code>	line feed
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\v</code>	vertical tab

Sometimes you have to deal with nonprinting whitespace characters. In these situations you probably will end up using the whitespace character class `\\s`. A common example is when you have to match tab characters, or line breaks.

The operating system Windows uses `\r\n` as an end-of-line marker. In contrast, Unix-like operating systems (including Mac OS) use `\n`.

Tab characters `\t` are commonly used as a field-separator for data files. But most text editors render them as whitespaces.

9.6 POSIX Character Classes

We finish this chapter with the introduction of another type of character classes known as **POSIX character classes**. These are yet another class construct that is supported by the regex engine in R.

Class	Description	Same as
<code>[:alnum:]</code>	any letter or digit	<code>[a-zA-Z0-9]</code>
<code>[:alpha:]</code>	any letter	<code>[a-zA-Z]</code>
<code>[:digit:]</code>	any digit	<code>[0-9]</code>
<code>[:lower:]</code>	any lower case letter	<code>[a-z]</code>
<code>[:upper:]</code>	any upper case letter	<code>[A-Z]</code>
<code>[:space:]</code>	any whitespace including space	<code>[\f\n\r\t\v]</code>
<code>[:punct:]</code>	any punctuation symbol	
<code>[:print:]</code>	any printable character	
<code>[:graph:]</code>	any printable character excluding space	
<code>[:xdigit:]</code>	any hexadecimal digit	<code>[a-fA-F0-9]</code>
<code>[:cntrl:]</code>	ASCII control characters	

Notice that a POSIX character class is formed by an opening bracket `[`, followed by a colon `:`, followed by some keyword, followed by another colon `:`, and finally a closing bracket `]`.

In order to use them in R, you have to wrap a POSIX class inside a regex character class. That is, you have to surround a POSIX class with brackets.

Once again, refer to the `pnx` vector to illustrate the use of POSIX classes:

```
pnx <- c('pan', 'pen', 'pin', 'p0n', 'p.n', 'p1n', 'paun')
```

Let's start with the `[:alpha:]` class, and see what does it match in `pnx`:

```
str_view(pnx, "[[:alpha:]]")
```

Now let's test it with `[:digit:]`

```
str_view(pnx, "[[:digit:]]")
```

Here's another 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
str_replace_all(la_vie, pattern="[:blank:]", replacement='')
```

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

```
# remove digits
str_replace_all(la_vie, pattern="[:punct:]", replacement='')
```

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

```
# remove digits
str_replace_all(la_vie, pattern="[:xdigit:]", replacement='')
```

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

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

```
[1] "\n\t"
```

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

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

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

```
[1] " \n \t "
```

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

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

10 Anchors

In this chapter, we discuss the regex topics known as *anchors* and *quantifiers*.

```
# packages used in this chapter
library(stringr)
```

10.1 Anchors

Anchors are metacharacters that help us assert the position, say, the beginning or end of the string.

Anchor	Description	Example
<code>^</code>	Matches a line starting with the substring.	<code>^New</code>
<code>\\\$</code>	Matches a line ending with the substring.	<code>y\$</code>
<code>^ \\\$</code>	Matches a line that starts and ends with substring, i.e., exact match.	<code>^Hi There\$</code>
<code>\\A</code>	Matches input starting with the substring.	<code>\\AHello</code>
<code>\\Z</code> and <code>\\z</code>	Matches input ending with the substring. <code>\\Z</code> also matches if there is a newline after the substring.	<code>End\\Z</code>

As an example, we will consider a simple character vector `universities` containing names of some universities.

```
universities <- c(
  "University of California, Berkeley",
  "University of California, San Francisco",
  "San Francisco State University",
  "California State University")
```

```
universities
```

```
[1] "University of California, Berkeley"
[2] "University of California, San Francisco"
[3] "San Francisco State University"
[4] "California State University"
```

10.1.1 Start of String

Let's try to detect university names that begin with **University**. To do this, we use `str_detect()` from the "stringr" package. Obviously, we need to provide a useful regex pattern that looks for the word **University** at the beginning of the text. How do we do this? With the caret `^` metacharacter to match for a **starting anchor**, followed by the string we want to match: `"^University"`

```
str_detect(universities, "^University")
```

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

As you can tell, only the first and second elements in `universities` are being matched.

```
str_extract(universities, "^University")
```

```
[1] "University" "University" NA NA
```

```
universities[str_detect(universities, "^University")]
```

```
[1] "University of California, Berkeley"
[2] "University of California, San Francisco"
```

10.1.2 End of String

Now let's try to detect university names that **end** with the word **University**. To do this, we have to use the metacharacter `$` to indicate the *ending anchor*, forming the pattern: `"University$"`.

```
str_detect(universities, regex("University$"))
```

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

Compared to the previous example, now only the third and fourth elements in `universities` are being matched.

```
str_extract(universities, "University$")
```

```
[1] NA NA "University" "University"
```

```
universities[str_detect(universities, "University$")]
```

```
[1] "San Francisco State University" "California State University"
```

To make things more interesting, we have modified the content of vector `universities`, now consisting of multiple lines (notice the newline characters `\`).

```
universities <- c(
  "University of California, Berkeley
  \nUniversity of California, San Francisco
  \nSan Francisco State University
  \nCalifornia State University\n")

cat(universities)
```

```
University of California, Berkeley
```

```
University of California, San Francisco
```

```
San Francisco State University
```

```
California State University
```

Say we are interested in extracting university names that end with the word `University`. We can try using `str_extract()` with the pattern `University\\$`

```
str_detect(universities, "University$")
```

```
[1] TRUE
```

```
universities[str_detect(universities, "University$")]
```

```
[1] "University of California, Berkeley\n  \nUniversity of California, San Francisco\n  \nSan Francisco State University\n  \nCalifornia State University"
```

Something was match, but what exactly? `str_view()` can give us the answer to this question:

```
str_view(universities, "University$")
```

Notice that the matched effectively occurred at the very end of the string in `universities`. But what if what we are really interested in is in matching the names `San Francisco State University` as well as `California State University`?

We shall use `str_extract_all()` instead of `str_extract()` to extract all occurrences of the pattern. In addition, the pattern should be specified inside the `regex()` function, using its `multiline` argument to tell R to expect input consisting of multiple lines. Here's how:

```
str_extract_all(universities,
                regex("[A-z]*University$", multiline = TRUE))
```

```
[[1]]
```

```
[1] "San Francisco State University" "California State University"
```

Lastly, let's try to extract the last word of our input from previous example.

Using `str_extract()` or `str_extract_all()` does not matter anymore. While we get a single output for both, the former returns a list and the latter returns a list of lists.

```
str_extract(universities, regex("[A-z]+\\Z", multiline = TRUE))
```

```
[1] "University"
```

Notice that `\\Z` works even in presence of a terminating newline `\n`. However, when we use `\\z`, this won't work until we remove the terminating `\n`.


```
str_extract(universities, regex("[A-z ]+\\z", multiline = TRUE))
```

```
[1] NA
```

With the newline terminator removed from the input, `\\z` works just as well.

```
universities <- c(
  "University of Southern California
  \nCalifornia State University
  \nStanford University
  \nUniversity of California, Berkeley")

str_extract(universities, regex("[A-z ]+\\Z", multiline = TRUE))
```

```
[1] " Berkeley"
```

```
str_extract(universities, regex("[A-z ]+\\z", multiline = TRUE))
```

```
[1] " Berkeley"
```

11 Quantifiers

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

11.1 Quantifier Metacharacters

As the name indicates, 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. The quantifier should be placed after the character, group or character class that is being quantified, denoted as *c* in the table below.

Quantifier	Description
<code>c?</code>	The preceding item is optional and will be matched at most once
<code>c*</code>	The preceding item will be matched zero or more times
<code>c+</code>	The preceding item will be matched one or more times
<code>c{n}</code>	The preceding item is matched exactly <i>n</i> times
<code>c{n,}</code>	The preceding item is matched <i>n</i> or more times
<code>c{n,m}</code>	The preceding item is matched at least <i>n</i> times, but no more than <i>m</i> times

For illustration purposes, let's create a vector `people`

```
people <- c(
  "rori",
  "emilia",
  "matteo",
  "mehmet",
  "filipe",
  "ana",
  "victoria")
```

```
people
```

```
[1] "rori"      "emilia"    "matteo"    "mehmet"    "filipe"    "ana"       "victoria"
```

We start with a simple example extracting all those names that contain at least five characters but no more than 7 characters. To do this, we define a pattern formed by the start anchor `^`, followed by a range of upper and lower case letters `[A-z]`, followed by the repetition pattern `{5,7}`, followed by the end anchor `$`

```
str_extract(people, "^[A-z]{5,7}$")
```

```
[1] NA          "emilia" "matteo" "mehmet" "filipe" NA          NA
```

The reason why we use anchors `^` and `$` is to make sure that we have an exact match.

Let's try to detect names of those individuals with one or more `a` or `e`.

```
str_detect(people, "[ae]+")
```

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

```
people[str_detect(people, "[ae]+")]
```

```
[1] "emilia"    "matteo"    "mehmet"    "filipe"    "ana"       "victoria"
```

In the last example, if we want to extract names that contain `a` or `e` we could follow this simple implementation. Points to note here:

- Character set `[ae]` could appear 1 or more times so we use the quantifier `+`.
- `.*` matches 0 or any number of characters where `.` is a wildcard dot and `*` represents the quantifier 0 or many `.`
- Pattern `.*[ae]+.*` looks for 1 or more occurrences of `[ae]` that can be preceded/followed by any number of other characters.

```
people <- c(
  "rori",
  "emilia",
  "matteo",
  "mehmet",
  "filipe",
  "ana",
  "victoria")

str_extract(people, regex(".*[ae]+.*"))
```

```
[1] NA          "emilia"    "matteo"    "mehmet"    "filipe"    "ana"       "victoria"
```

11.1.1 What do groups mean in Regex?

We visited character classes in one of the sections. For situations where we would like to group character classes or regex pattern before using a quantifier, we indicate grouping using paranthesis.

Consider an example where we would like to extract only strings with two names separated by a whitespace. For illustrative purpose, the strings end with a whitespace.

```
people <- c(
  "rori rholfs",
  "emilia huerta ",
  "matteo fumagalli ",
  "mehmet ",
  "filipe vieira",
  "ana chen",
  "victoria kim ")

str_extract(people, "([A-z]+[ ]){2}")
```

```
[1] NA          "emilia huerta "    "matteo fumagalli "
[4] NA          NA                  NA
[7] "victoria kim "
```

We could also use pre-built class `[[:alpha:]]` in the above example.

```
str_extract(people, regex("([[:alpha:]]+[ ]){2}"))
```

```
[1] NA "emilia huerta " "matteo fumagalli "
[4] NA NA NA
[7] "victoria kim "
```

11.2 Greedy vs Lazy Match

As you might have noted in previous cases, regex tend to return *greedy* results., i.e., the longest match possible for a given expression. Let's explore this idea further and see if we can force it to be *lazy*.

Consider again one of our previous examples, but this time extracting those names that contain at least four characters but no more than 6 characters:

```
people <- c(
  "rori",
  "emilia",
  "matteo",
  "mehmet",
  "filipe",
  "ana",
  "victoria")

str_extract(people, "[A-z]{4,6}$")
```

```
[1] "rori" "emilia" "matteo" "mehmet" "filipe" NA NA
```

The quantifier `{4,6}` returned a *greedy* match, i.e., the result was of the maximum length possible.

Let us remove the anchors to see whether it is indeed *greedy*. By removing the anchors, it prints the first 4-6 characters of all names, provided name length is at the minimum 4.

```
str_extract(people, regex("[A-z]{4,6}"))
```

```
[1] "rori" "emilia" "matteo" "mehmet" "filipe" NA "victor"
```

We could make it *lazy* by adding a `?` after the quantifier. For names `emilia`, `matteo`, `mehmet`, `filipe` and `victoria`, it prints only the first 4 characters.

```
str_extract(people, regex("[A-z]{4,6}?"))
```

```
[1] "rori" "emil" "matt" "mehm" "fili" NA      "vict"
```

Similarly, we could make other quantifiers *lazy*.

Original Quantifier (Greedy)	Lazy Version
<code>c?</code>	<code>c??</code>
<code>c*</code>	<code>c*?</code>
<code>c+</code>	<code>c+?</code>
<code>c{n}</code>	<code>c{n}?</code>
<code>c{n,}</code>	<code>c{n,}?</code>
<code>c{n,m}</code>	<code>c{n,m}?</code>

12 Boundaries and Look Arounds

In [chapter 9](#) we introduced a handful of character classes such as `\d` which matches any digit, or `\w` which matches any character considered to be part of a word. Among these classes, there are two special metacharacters, `\b` and `\B`, known as **boundaries**, that deserve further discussion.

Likewise, we need to talk about another useful regex concept known as **look arounds**. The patterns behind this notion will allow us to match tokens *around* auxiliary tokens are not to be matched.

12.1 Boundaries

Boundaries are metacharacters too and match based on what precedes or follows current position.

Boundary	Description	Example
<code>\\b</code>	Matches a word boundary, i.e., when a side is not <code>[A-z0-9_]</code>	<code>\\bHi \\bHi\\b</code>
<code>\\B</code>	Matches when not a word boundary, i.e., when a side is <code>[A-z0-9_]</code>	<code>\\BHi</code>

To understand word boundaries, let us go over a simple example. Consider the string in vector `book` shown below

```
book <- c("This book is an irresistible thesis")
book
```

```
[1] "This book is an irresistible thesis"
```

Suppose we are interested in matching the pattern `is`.

If you observe the text in `book`, you'll notice that `is` appears in the words `This`, `is`, `irresistible`, and `this`. Depending on which function you use, you will be able to match just the first occurrence

```
# matching first occurrence
str_view(book, "is")
```

or match all occurrences:

```
# matching all occurrences
str_view_all(book, "is")
```

Warning: ``str_view()`` was deprecated in `stringr 1.5.0`.
i Please use ``str_view_all()`` instead.

Instead of matching `is` without any other restriction or condition, we can also think of three additional matching cases that are more concrete:

- 1) Extract words that exactly match with `is`
- 2) Extract words that contain `is` in between characters
- 3) Extract words that end with `is`

Case 1: Words that exactly match `is`

To match (or extract) words that exactly match the word `is`, we use the boundary-word pattern `\\b`. Note the optional use of `ignore_case` argument to `regex()`

```
str_view_all(
  string = book,
  pattern = regex("\\bis\\b", ignore_case = TRUE))
```

Case 2: Words containing `is` in between characters

To match words that contain `is` in between characters, such as “irresistible”, we use the *not a word boundary* `\\B` on both sides of `is`:


```
str_view_all(
  string = book,
  pattern = regex("\\Bis\\B", ignore_case = TRUE))
```

To extract the entire word “irresistible”, we must include `[A-z]*` on either side of `\\Bis\\B`, otherwise the pattern `\\Bis\\B` will only match `is`

```
str_extract(
  string = book,
  pattern = regex("[A-z]*\\Bis\\B[A-z]*", ignore_case = TRUE))
```

```
[1] "irresistible"
```

Notice that we use `[A-z]*` instead of `[A-z]+` to specifically showcase that no other `is` got matched as `*` denotes 0 or any.

You may ask “What if we don’t surround `\\Bis\\B` with `[A-z]*`?” Here is what happens:

```
str_view_all(
  string = book,
  pattern = regex("[A-z]*\\Bis", ignore_case = TRUE))
```

The pattern `[A-z]*\\Bis` matches “This”, “irresis”, and “thesis”, which are extracted as follows:

```
str_extract_all(
  string = book,
  pattern = regex("[A-z]*\\Bis", ignore_case = TRUE))
```

```
[[1]]
[1] "This"      "irresis" "thesis"
```

Case 3: Match (or extract) words that end with `is`

If you are interested in extracting words that end with `is`, then you need to use a pattern with the word boundary `is\\b`. Now, you also need to determine if `is` should be part of a word with one or more preceding word-characters, or if `is` should not be preceded by any word-characters.

```
# end with "is", with preceding word-characters
str_view_all(
  string = book,
  pattern = regex("[A-z]+is\\b", ignore_case = TRUE))
```

```
# end with "is", without preceding word-characters
str_view_all(
  string = book,
  pattern = regex("\\bis\\b", ignore_case = TRUE))
```

12.2 Look Arounds

As the name suggests, this type of pattern allows us to *look around* the string in order to match the desired pattern. To be more precise, *look arounds* indicate positions just like anchors, `$` and `^`.

There are four types of look arounds, listed in the following table.

Look Around	Notation	Description
Positive Look Ahead	<code>A(?:=pattern)</code>	Check if pattern follows A
Negative Look Ahead	<code>A(?:!pattern)</code>	Check if pattern does not follow A
Positive Look Behind	<code>(?<=pattern)A</code>	Check if pattern precedes A
Negative Look Behind	<code>(?<!pattern)A</code>	Check if pattern does not precede A

In this table, A refers to a character set or group that we are trying to match.

12.2.1 Look Aheads

Let us look at some examples for Look Aheads. To do this, consider the vector `heights` shown below

```
heights <- c("40cm", "23", "60cm", "57", "133cm")
heights
```

```
[1] "40cm" "23"    "60cm" "57"    "133cm"
```

Suppose we want to extract the heights without the unit of measurement, that is, we want to obtain the numeric values but not the letters cm. We can do this by specifying a pattern to match one or more digits `[0-9]+`

```
str_extract(heights, "[0-9]+")
```

```
[1] "40" "23" "60" "57" "133"
```

Let's change the format of `heights` by collapsing all of its elements into a single string, with height values separated by commas:

```
heights <- paste(heights, collapse = ", ")
heights
```

```
[1] "40cm, 23, 60cm, 57, 133cm"
```

With this modified `heights`, if we use the previous command, we only extract the first occurrence:

```
str_extract(heights, "[0-9]+")
```

```
[1] "40"
```

In order to extract *all* occurrences, we must use `str_extract_all()`

```
str_extract_all(heights, "[0-9]+")
```

```
[[1]]
[1] "40" "23" "60" "57" "133"
```

Let's change our input to contain heights that are in inches

```
heights <- "40cm, 23in, 60cm, 57, 133cm, 15in, 99"
heights
```

```
[1] "40cm, 23in, 60cm, 57, 133cm, 15in, 99"
```

and reapply the previous command

```
str_extract_all(heights, "[0-9]+")
```

```
[[1]]  
[1] "40" "23" "60" "57" "133" "15" "99"
```

In case we want to retrieve only those heights with unit of measurement `cm`, we could use a *positive look ahead*. In the syntax `A(?=pattern)`, the pattern we look for would be `cm`. Any number that is followed by `cm` will be extracted hence `A` is `[0-9][0-9]+`.

Don't forget that `(?=cm)` does not extract `cm`, it is used to assert position only!

```
heights <- "40cm, 23in, 60cm, 57, 133cm, 15in, 99"  
  
str_extract_all(heights, "[0-9][0-9]+(?=cm)")
```

```
[[1]]  
[1] "40" "60" "133"
```

In case we want to extract all heights that don't have `cm` as the unit of measurement, we could use *negative look ahead*. In the syntax `A(?!pattern)`, the pattern here is `cm` and `A` should be character class `[0-9]` with quantifier `+` (one or many).

```
heights <- "40cm, 23in, 60cm, 57, 133cm, 15in, 99"  
  
str_extract_all(heights, "[0-9][0-9]+(?!cm)")
```

```
[[1]]  
[1] "23" "57" "13" "15" "99"
```

Similarly, using *negative look ahead* to extract all heights that don't have `cm` as the unit of measurement should work too, but in the code snippet below, we get an incorrect output. Can you guess why?

```
heights <- "40cm, 23in, 60cm, 57, 133cm, 15in, 99"  
  
str_extract_all(heights, regex("[0-9]+(?!cm)"))
```

```
[[1]]  
[1] "4" "23" "6" "57" "13" "15" "99"
```

This is incorrect since we extract the 4 from 40cm, 6 from 60cm, and 13 from 133cm additional to our actual answer.

We could overcome this by specifically mentioning that the number (height) **cannot** be followed by:

- the pattern `cm`
- some number (i.e, from 40cm, we should not extract 4)

To do this, we could use **alternation**, denoted as pipe `|`, which is similar to **OR**.

The pattern in `A(?!pattern)` is now `cm` or `[0-9]+`, which we can represent as `(cm|[0-9]+)`. Note that our pattern is enclosed within parenthesis.

```
heights <- "40cm, 23in, 60cm, 57, 133cm, 15in, 99"  
  
str_extract_all(heights, regex("[0-9]+(?!(cm|[0-9]+))"))
```

```
[[1]]  
[1] "23" "57" "15" "99"
```

The illustration above is typical of regular expressions. As the test cases become complex, you will have to tweek the expression to include all the corner cases.

12.2.2 Look Behinds

As the name suggests, this type of metacharacters allows us to *look behind* the current position for presence or absence of a pattern. This works the same way as look ahead, except that we look for the preceding characters.

Let us look at some examples.

Consider a vector `courses` that has the name and year of some statistics courses over different semesters.

```
courses <- c(  
  "Stat_133 2020",  
  "Stat_154 2020",  
  "Stat_133 2019",  
  "Stat_151 2018",
```

```
"Stat_151 2020",
"Stat_154 2018")
```

```
courses
```

```
[1] "Stat_133 2020" "Stat_154 2020" "Stat_133 2019" "Stat_151 2018"
[5] "Stat_151 2020" "Stat_154 2018"
```

We will use a *positive look-behind* to extract the years associated to `Stat_133`. In the syntax `(?<=pattern)P`, `pattern` is `(Stat_133)` and `P` is the semester we want to extract, i.e., `[0-9]{4}`.

```
courses <- c(
  "Stat_133 2020",
  "Stat_154 2020",
  "Stat_133 2019",
  "Stat_151 2018",
  "Stat_151 2020",
  "Stat_154 2018")

str_extract(courses, regex("(?<=(Stat_133 ))[0-9]{4}"))
```

```
[1] "2020" NA      "2019" NA      NA      NA
```

Similarly, we can use a *negative look-behind* to extract semesters of courses that are not `Stat_133`. In the syntax `(?<!(pattern)P)`, `P` is `[0-9]{4}` and the pattern is `(Stat_133)`.

```
courses <- c(
  "Stat_133 2020",
  "Stat_154 2020",
  "Stat_133 2019",
  "Stat_151 2018",
  "Stat_151 2020",
  "Stat_154 2018")

str_extract(courses, "(?<!(Stat_133 ))[0-9]{4}")
```

```
[1] NA      "2020" NA      "2018" "2020" "2018"
```

12.3 Logical Operators in Regex

We don't have earmarked Logical Operators in Regex, however, a few syntaxes could replicate these.

We saw in one of the examples for look aheads that logical OR is expressed using the pipe symbol `|` in regex. This is also known as 'Alternation' operation.

There is no AND in regex, but it can be synthesized using look arounds. For NOT operation, the `^` symbol works in character classes but cannot be used for groups as explained later.

Operation	Syntax	Example
OR	Pipe symbol <code> </code>	<code>pattern1 pattern2</code>
NOT	Cap symbol <code>^</code>	<code>[^aeiou]</code>
AND	Synthetic AND	<code>(?=P1)(?=P2)</code>

12.3.1 Logical OR

Consider the vector `people` from a previous example.

```
people <- c(
  "rori",
  "emilia",
  "matteo",
  "mehmet",
  "filipe",
  "ana",
  "victoria")
```

```
people
```

```
[1] "rori"      "emilia"    "matteo"    "mehmet"    "filipe"    "ana"       "victoria"
```

If we want to display names that are of length 3 OR of length 4, we use the pipe symbol.

```
str_extract(people, regex("^[A-z]{3}|[A-z]{4}$"))
```

```
[1] "rori" NA      NA      NA      "ana" NA
```

Note that for the above example, we could use quantifier {3,4} and still obtain the same result.

```
str_extract(people, regex("^[A-z]{3,4}$"))
```

```
[1] "rori" NA      NA      NA      "ana" NA
```

12.3.2 Logical NOT

Using the NOT `^`, we could extract names that don't contain `e` or `u`.

```
str_extract(people, regex("^[^eu]+$"))
```

```
[1] "rori"      NA      NA      NA      NA      "ana"      "victoria"
```

Note that `^` operation cannot be used for groups. This is because it is unclear in such cases if we are using `^` as a NOT operator or an anchor. In such cases we could use a negative look around. Let's say, we have a group `(ia)` and we want to extract names without the group `(ia)`.

```
str_extract(people, regex("^(?!.*ia).*"))
```

```
[1] "rori"      NA      "matteo" "mehmet" "filipe" "ana"      NA
```

12.3.3 Logical AND

Consider a case where we want to extract names with length greater than 4 and containing letter `o`. We have two conditions and we need to AND them.

- Condition 1: Length > 4, the pattern is `(?=. {5,})`
- Condition 2: Contains `o`, the pattern is `(?=.*o)`

The two conditions can be used together.

```
people <- c(
  "rori",
  "emilia",
  "matteo",
```



```
"mehmet",  
"filipe",  
"ana",  
"victoria")
```

```
str_extract(people, regex("(?=. {5,})(?=. *o).*"))
```

```
[1] NA      NA      "matteo" NA      NA      NA      "victoria"
```

13 Regex Functions in "stringr"

In the previous chapters we talked about regular expressions in general; we discussed the particular way in which R works with regex patterns; and we quickly presented some functions to manipulate strings with regular expressions. In this chapter we are going to describe in more detail the functions for regular expressions available in both the "stringr" package.

As you know, we have already presented some of the functions in the R package "stringr" for regular expressions. As we mentioned, all these 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 occurrence, we should use **str_locate()**; to locate the positions of all matches we should use **str_locate_all()**.

13.1 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 wrapper 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 matches
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
```

13.2 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 tell, 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"`.

13.3 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`.

13.4 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.

13.5 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]]
      [,1]

```

Compared to `str_match()`, the output of `str_match_all()` is a **list**. Note 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`.

13.6 Locate first match with `str_locate()`

Besides detecting, extracting and matching regex patterns, "**stringr**" allows us to *locate* occurrences of patterns. For locating the position of the **first** occurrence 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.

13.7 Locate all matches with `str_locate_all()`

To locate not just the first but all the occurrence 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.

13.8 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"
```

13.9 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;"
```

13.10 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 individual 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"
```

13.11 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"      ""
```

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"      ""       ""
```

14 Regex Functions in R

In the previous chapters we talked about regex functions available in the package `"stringr"`. In this chapter we are going to describe more regular expression functions but this time from the `"base"` package (i.e. native regex functions in R).

14.1 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.

14.1.1 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 tell 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 unmatched 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.

14.1.2 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`.

14.1.3 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
attr(,"match.length")
[1] 3 -1 -1 3
```

```
attr(,"index.type")
[1] "chars"
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.

14.1.4 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.

```
# 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(,"index.type")
[1] "chars"
attr(,"useBytes")
```



```

[1] TRUE

[[2]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

[[3]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

[[4]]
[1] 11
attr(,"match.length")
[1] 3
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

```

14.1.5 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
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

```
[[2]]
[1] -1
attr("match.length")
[1] -1
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

```
[[3]]
[1] -1
attr("match.length")
[1] -1
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

```
[[4]]
[1] 11
attr("match.length")
[1] 3
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

14.2 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)
```

14.2.1 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 you 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.

14.2.2 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* or *global* substitution). 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 occurrence of **R** in the word **FREE** is taken into account and `gsub()` changes it to **FRREE**.

14.3 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"
```

Part IV

IV Applications

15 Fun Plots (part 1)

This chapter and the next one are dedicated to create a couple of fun projects in which you get to apply what we have covered so far.

The idea is to produce some plots with some text on it. But not any kind of text.

15.1 Me & You Plot

The first plot is intended to be a postcard—e.g. for Saint Valentine’s day. After reading this kind of tutorial, you should be able to make your own plot, print it and give it to your significant other.

The idea is to make a chart with your name and the name of your significant other, adding a touch of randomness in the location of the text, the sizes, and the colors.

First we generate the x-y coordinates. We’ll use 100 points, and set the random seed to 333:

```
# random seed
set.seed(333)

# x-y coordinates
n <- 100
x <- rnorm(n)
y <- rnorm(n, 1, 2)
```

The first step involves producing a very basic raw plot (nothing fancy). We use `plot()` for this purpose:

```
plot(x, y)
```

The `plot()` produces a scatter diagram with a 100 points on it.

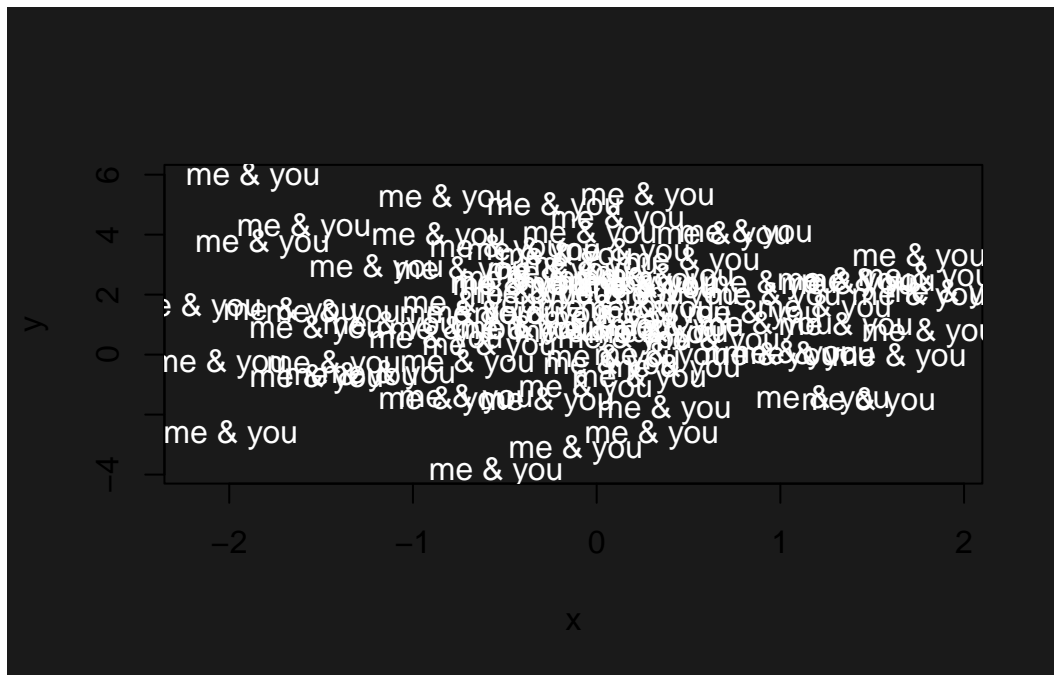
The following step consists of replacing the dots by some text: your name and the one of your significant other. To hide the dots, we set the parameter `type = "n"`, which means that we don’t want anything to be plotted. To show the text, we use the low level plotting function `text()`. We use the same coordinates, but this time we specify the displayed `labels`:

```
plot(x, y, type = "n")
text(x, y, labels = "me & you")
```

Again, this is a very preliminary plot; something basic that allows us to start getting a feeling of how the chart looks like.

The second step is to change the background color. One way to do this is by specifying the `bg` graphical parameter inside the `par()` function:

```
# graphical parameters
op <- par(bg = "gray10")
# plot text
plot(x, y, type = "n")
text(x, y,
     labels = "me & you",
     col = "white")
```



```
# reset default parameters
par(op)
```

`par()` has default settings. Everytime you call `par()` and change one of the associated parameters, the subsequent plots will be displayed with those values. To set parameters in a

temporary way, you can assign them to an object: i.e. `op`. After the plot is produced, we reset the default graphical parameters with the instruction `par(op)`.

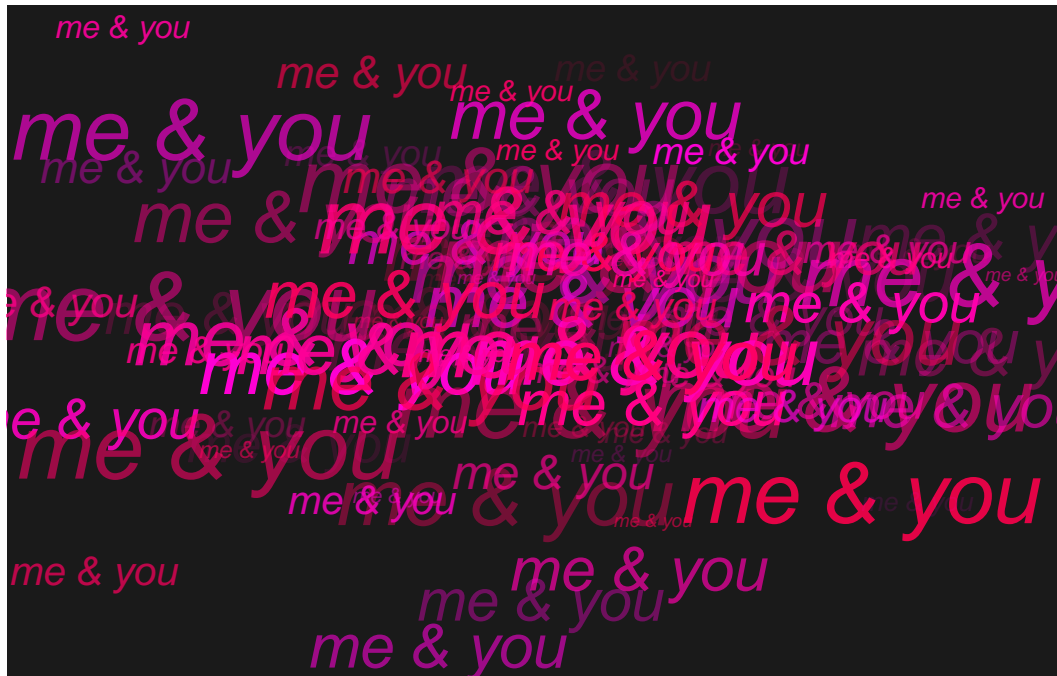
We are getting closer to the desired look of the postcard. The final stage is to add some color to the text, and change their size. The size of the labels will also be random with a uniform distribution.

R provides several ways to specify colors. In this example we will use the `hsv()` function (i.e. hue-saturation-value). This function requires three parameters: hue (color), saturation, and value. Hues are specified with a range from 0 to 1. We generate some random numbers in the interval 0.85 - 0.95 to get some hues red, pink, fuchsia colors. `hsv()` also takes the optional parameters `alpha` to determined the alpha transparency.

```
# text size
sizes <- runif(n, 0.5, 3)

# text color
hues <- runif(n, 0.85, 0.95)
alphas <- runif(n, 0.1, 1)

op <- par(bg = "gray10", mar = rep(0, 4))
plot(x, y, type = "n", axes = FALSE,
      xlab = '', ylab = '')
text(x, y,
      labels = "me & you",
      font = 3,
      col = hsv(hues, 1, 1, alphas),
      cex = sizes)
```



```
par(op)
```

To save the image, we call `pdf()`. To give the image the dimensions of a standard postcard, you can specify `width = 5` and `height = 3.5`, that is, 5 inches wide and 3.5 inches tall (you can choose other dimensions if you want):

```
pdf(file = "figure/me-and-you3.pdf", width = 5, height = 3.5)
op <- par(bg = "gray10",
          mar = rep(0, 4))
plot(x, y, type = "n", axes = FALSE,
     xlab = '', ylab = '')
text(x, y,
     labels = "me & you",
     font = 3,
     col = hsv(hues, 1, 1, alphas),
     cex = sizes)
par(op)
dev.off()
```

If you save the image in png format, you could also use it as a wallpaper for your computer:



Figure 15.1: A wallpaper chart for your significant other

16 Fun Plots (part 2)

In the preceding chapter we created a first entertaining plot. In this chapter we move on to our second fun graphic involving some strings.

16.1 Colored Jittery Text

For our second fun plot, let's consider some text. For instance, part of the famous speech by Dr. King:

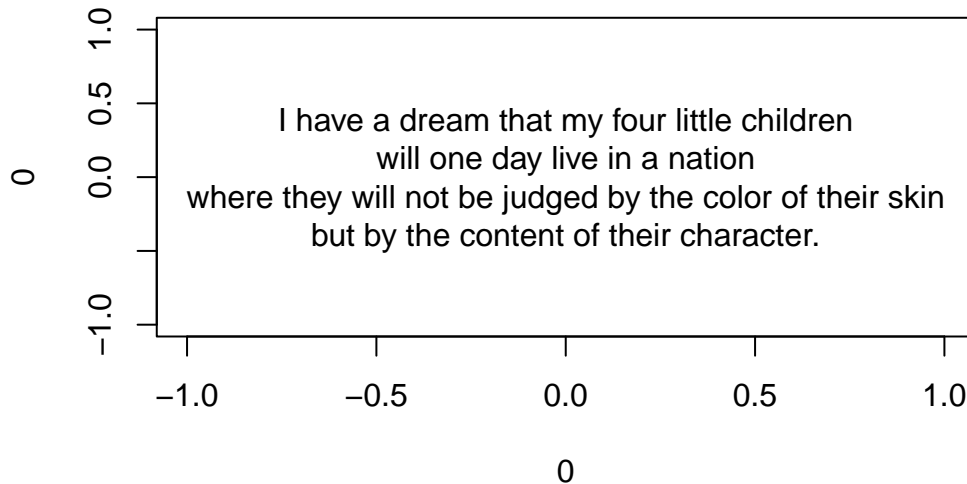
I have a dream that my four little children will one day live in a nation, where they will not be judged by the color of their skin but by the content of their character.

Let's put the text in a character vector:

```
speech <- c(
  "I have a dream that my four little children",
  "will one day live in a nation",
  "where they will not be judged by the color of their skin",
  "but by the content of their character."
)
```

First we are going to plot the text as a single string, collapsing it with some new-line characters `"\n"`:

```
plot(0, 0, type = 'n')
text(0, 0, paste(speech, collapse = '\n'))
```



Notice that we're using basic coordinates with a center for the plot on (0,0). This helps us identify reference points both in the x-axis and the y-axis for future manipulation of the character strings.

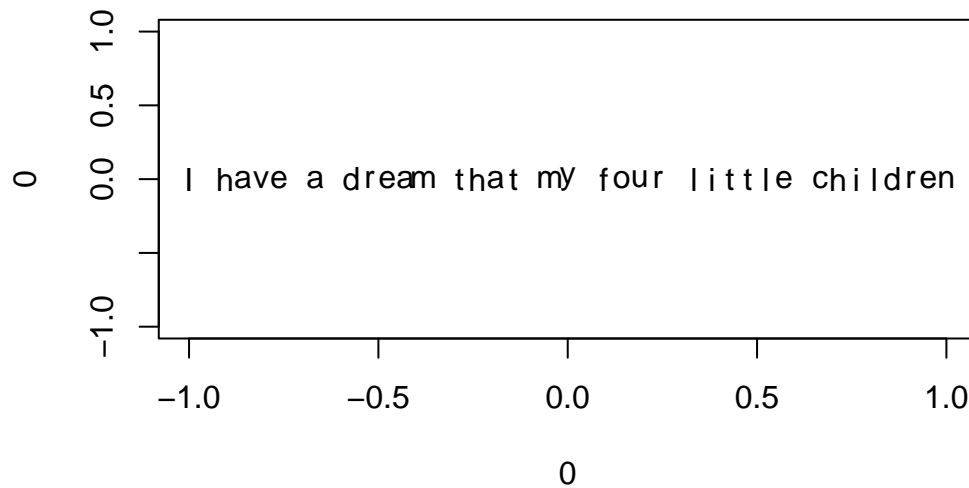
In order to plot each character with a different size and with different coordinates, we need to split the strings into individual characters. To do this we use the almighty function `strsplit()`. For testing purposes, let's take one sentence from MLK's speech. Also, we'll assign x-axis coordinates by dividing the range of the x-axis (from -1 to 1) in equal parts taking into account the number of characters in the input string:

```
# one sentence for testing purposes
str <- "I have a dream that my four little children"

# splitting str in individual characters
txt <- unlist(strsplit(str, split = ''))

# x-coordinates for each character
nchars <- length(txt)
xs <- seq(-1, 1, length.out = nchars)

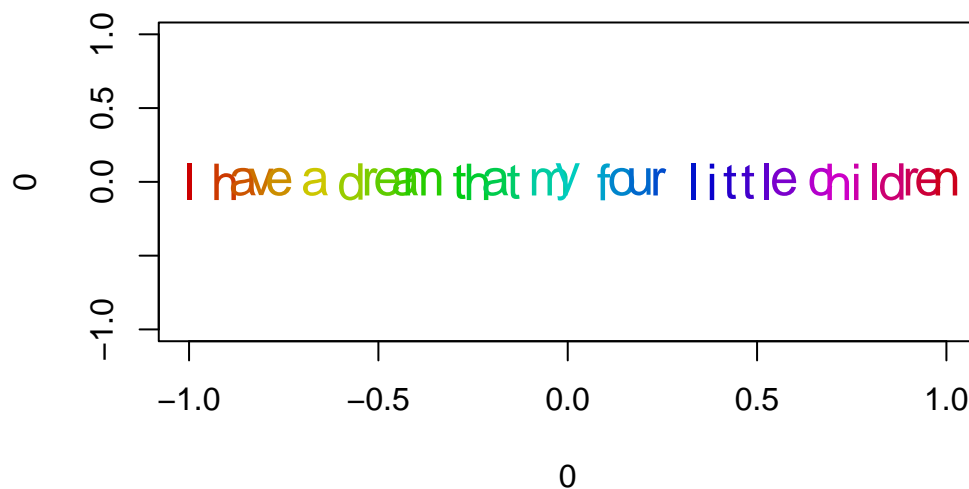
plot(0, 0, type = 'n')
for (i in 1:nchars) {
  text(xs[i], 0, labels = txt[i], cex = 1)
}
```



Now we can add color and modify the size of the characters. In this case we'll use the `rainbow()` palette for colors, and the `runif()` function to generate uniform values to modify the sizes of the letters:

```
# character expansion and colors
rcex <- runif(nchars, 1.5, 2)
cols <- rainbow(nchars, v = 0.8)

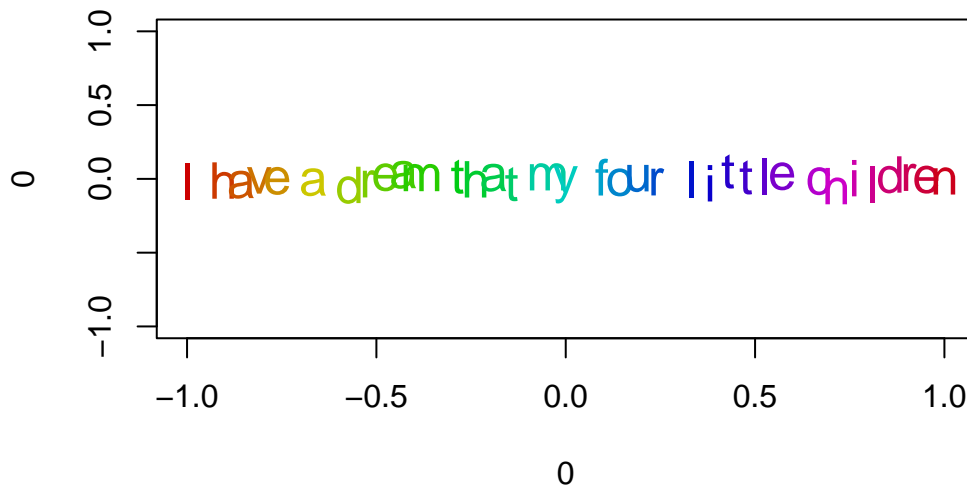
plot(0, 0, type = 'n')
for (i in 1:nchars) {
  text(xs[i], 0, labels = txt[i], cex = rcex, col = cols[i])
}
```



Finally, let's add some jitter to the y-position. We do this by adding some noise following a normal distribution with mean zero and a small standard deviation via the `rnorm()` function:

```
# jitter for y-coordinates
ry <- rnorm(nchars, 0, 0.03)

plot(0, 0, type = 'n')
for (i in 1:nchars) {
  text(xs[i], ry[i], labels = txt[i], cex = rcex, col = cols[i])
}
```



16.1.1 Assembling the plot

Once we have all the necessary elements, we can add the rest of the speech lines, remove the axes, and add a little bit of color to the background. Moreover, to manipulate each line of text (i.e. each element in the `speech` vector) we break them down with `sapply()`, so we can loop over each line inside the plot:

```
strs <- sapply(speech, function(x) strsplit(x, split=''))
ys <- seq(0.75, -0.75, length.out = length(strs))

# random seed
set.seed(34587)

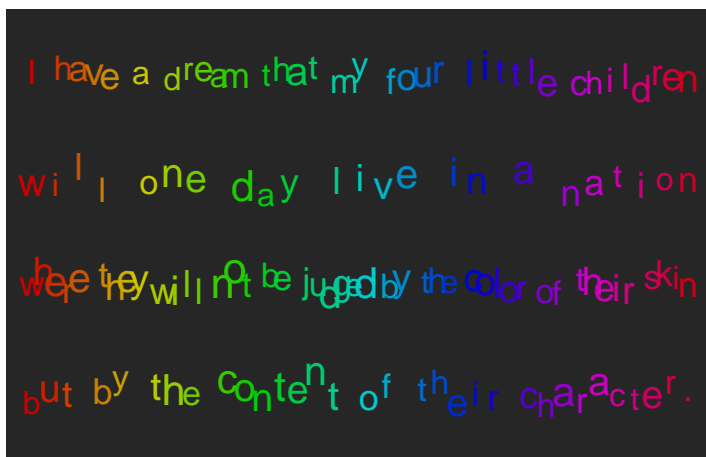
# plot text
op <- par(mar = rep(0, 4), bg = 'gray15')
plot(0, 0, type = 'n', axes = FALSE)
```

```

for (elem in 1:length(strs)) {
  s <- unlist(strs[elem])
  ns <- length(s)
  xs <- seq(-1, 1, length.out = ns)
  rcex <- runif(ns, 1.5, 2)
  cols <- rainbow(ns, v = 0.8)
  ry <- rnorm(ns, 0, 0.03)

  for (i in 1:ns) {
    text(xs[i], ys[elem] + ry[i], labels = s[i],
         cex = rcex[i], col = cols[i])
  }
}
par(op)

```



I have a dream that my four little children
 will one day live in a nation
 where they will not be judged by the color of their skin
 but by the content of their character.

17 Basic Examples

This chapter provides more elaborated examples than the simple demos presented so far. The idea is to show you less abstract scenarios and cases where you could apply the functions and concepts covered so far.

17.1 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 others, 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.

17.2 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:

```
# 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:

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

```
[1] "gfedcba"
```

```
# try with non-character input
reverse_chars(12345)
```

Error in strsplit(string, split = ""): 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"
```

17.3 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"
```

17.4 Names of Files

Imagine that you need to generate the names of 10 data `.csv` files. All the files have the same prefix name but each of them has a different number: `file1.csv`, `file2.csv`, ... , `file10.csv`.

There are several ways in which you could generate a character vector with these names. One naive option is to manually type those names and form a vector with `c()`

```
c('file1.csv', 'file2.csv', 'file3.csv')
```

```
[1] "file1.csv" "file2.csv" "file3.csv"
```

But that's not very efficient. Just think about the time it would take you to create a vector with 100 files. A better alternative is to use the vectorized nature of `paste()`

```
paste('file', 1:10, '.csv', sep = "")
```

```
[1] "file1.csv" "file2.csv" "file3.csv" "file4.csv" "file5.csv"
[6] "file6.csv" "file7.csv" "file8.csv" "file9.csv" "file10.csv"
```

Or similarly with `paste0()`

```
paste0('file', 1:10, '.csv')
```

```
[1] "file1.csv" "file2.csv" "file3.csv" "file4.csv" "file5.csv"
[6] "file6.csv" "file7.csv" "file8.csv" "file9.csv" "file10.csv"
```

17.5 Valid Color Names

R comes with the function `colors()` that returns a vector with the names (in English) of 657 colors available in R. How would you write a function `is_color()` to test if a given name—in English—is a valid R color. If the provided name is a valid R color, `is_color()` should return `TRUE`. If the provided name is not a valid R color `is_color()` should return `FALSE`.

```
is_color <- function(x) {
  x %in% colors()
}
```

Lets test it:

```
is_color('yellow') # TRUE
```

```
[1] TRUE
```

```
is_color('blu')    # FALSE
```

```
[1] FALSE
```

```
is_color('turkuiose') # FALSE
```

```
[1] FALSE
```

Another possible way to write `is_color()` is comparing if `any()` element of `colors()` equals the provided name:

```
is_color2 <- function(x) {  
  any(colors() == x)  
}
```

Test it:

```
is_color2('yellow') # TRUE
```

```
[1] TRUE
```

```
is_color2('blu')    # FALSE
```

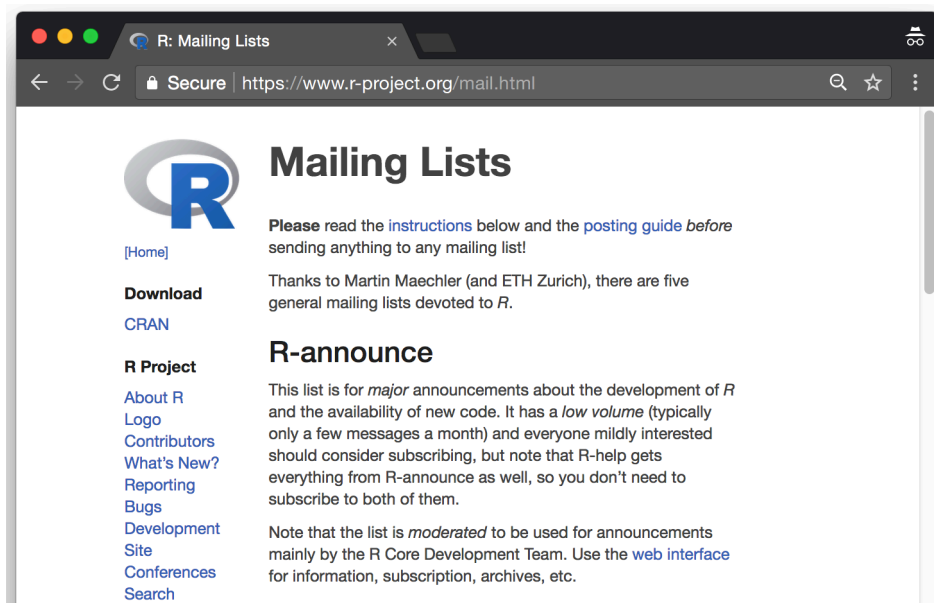
```
[1] FALSE
```

```
is_color2('turkuiose') # FALSE
```

```
[1] FALSE
```

18 Matching HTML Tags

In this chapter we review an example that deals with some basic handling of HTML tags. The data for this practical application is the webpage for the R mailing lists: <http://www.r-project.org/mail.html> (see screenshot below)



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

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

Additionally, there are several specific **Special Interest Group** (SIG) mailing lists. Here's a screenshot with some of the special groups:

Special Interest Groups

Additionally, there are several specific *Special Interest Group* (=: SIG) mailing lists; however do post to *only one* list at time ('SIG' or general one), cross-posting is considered to be impolite.

- **R-SIG-Mac**: R Special Interest Group on Mac ports of R
- **R-SIG-DB**: R SIG on Database Interfaces
- **R-SIG-Debian**: R Special Interest Group for Debian ports of R
- **R-SIG-dynamic-models**: Special Interest Group for Dynamic Simulation Models in R
- **R-SIG-ecology**: Using R in ecological data analysis
- **R-SIG-Epi**: R for epidemiological data analysis
- **R-SIG-Fedora**: R Special Interest Group for Fedora and Redhat ports of R
- **R-SIG-Finance**: Special Interest Group for 'R in Finance'
- **R-SIG-Geo**: R Special Interest Group on using Geographical data and Mapping
- **R-SIG-gR**: R SIG on gRaphical models
- **R-SIG-GUI**: R Special Interest Group on GUI Development
- **R-SIG-HPC**: R SIG on High-Performance Computing

18.1 Attributes href

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 (in just one line of code):

```
"<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-mac\">
<code>R-SIG-Mac</code></a>: R Special Interest Group on Mac ports of R</p></li>"

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

18.1.1 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")
```

In case you are having problem downloading the HTML file, you can also find a copy in the github repository for data sets of this book. You can use the code below to download a copy of the file to your working directory:

```
# download file
github <- "https://raw.githubusercontent.com/gastonstat/strings-data"
textfile <- "/main/data/mail.html"
download.file(url = paste0(github, textfile), destfile = "mail.html")
```

Once you have the data in your working directory, you can import in R with `readLines()`

```
mail_lists <- readLines("mail.html")
```

The first elements in `mail_lists` are:

```
head(mail_lists)
```

```
[1] "<!DOCTYPE html>"
[2] "<html lang=\"en\">"
[3] "  <head>"
[4] "    <meta charset=\"utf-8\">"
[5] "    <meta http-equiv=\"X-UA-Compatible\" content=\"IE=edge\">"
[6] "    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">"
```

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.

```
'^.*<p><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 quotation marks `' '`. Here is how we can get the SIG links:

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

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

# let's see first 5 elements
head(sig_hrefs, n = 5)
```

```
[1] "<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-mac\"><code>R-SIG-Mac</code>"
[2] "<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-db\"><code>R-SIG-DB</code>"
[3] "<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-debian\"><code>R-SIG-Debian</code>"
[4] "<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-dynamic-models\"><code>R-SIG-Dynamic-Models</code>"
[5] "<li><p><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-ecology\"><code>R-SIG-Ecology</code>"
```

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
sigs = sub(sig_pattern, '\\1', sig_hrefs)
sigs
```

```
[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-ecology"
[6] "https://stat.ethz.ch/mailman/listinfo/r-sig-epi"
[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-insurance"
[14] "https://stat.ethz.ch/mailman/listinfo/r-sig-jobs"
[15] "https://stat.ethz.ch/mailman/listinfo/r-sig-mediawiki"
[16] "https://stat.ethz.ch/mailman/listinfo/r-sig-meta-analysis"
[17] "https://stat.ethz.ch/mailman/listinfo/r-sig-mixed-models"
[18] "https://stat.ethz.ch/mailman/listinfo/r-sig-networks"
[19] "https://stat.ethz.ch/mailman/listinfo/r-sig-phylo"
[20] "https://stat.ethz.ch/mailman/listinfo/r-sig-qa"
[21] "https://stat.ethz.ch/mailman/listinfo/r-sig-robust"
[22] "https://stat.ethz.ch/mailman/listinfo/r-sig-teaching"
```

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.

19 Data Cleaning

In this application we are going to work with the Men's Long Jump World Record Progression data from wikipedia (see screenshot below).

[https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression#Low_altitude_record_prog](https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression#Low_altitude_record_progression)

Record progression [\[edit \]](#)

Mark	Wind	Athlete	Venue	Date
7.61 m (24 ft 11½ in)		 Peter O'Connor (IRE)	Dublin, Ireland	5 August 1901 ^[1]
7.69 m (25 ft 2¾ in)		 Edward Gourdin (USA)	Cambridge, United States	23 July 1921 ^[1]
7.76 m (25 ft 5½ in)		 Robert LeGendre (USA)	Paris, France	7 July 1924 ^[1]
7.89 m (25 ft 10¾ in)		 DeHart Hubbard (USA)	Chicago, United States	13 June 1925 ^[1]
7.90 m (25 ft 11 in)		 Edward Hamm (USA)	Cambridge, United States	7 July 1928 ^[1]
7.93 m (26 ft 0 in)	0.0	 Sylvio Cator (FRA)	Paris, France	9 September 1928 ^[1]
7.98 m (26 ft 2 in)	0.5	 Chuhei Nambu (JPN)	Tokyo, Japan	27 October 1931 ^[1]
8.13 m (26 ft 8 in)	1.5	 Jesse Owens (USA)	Ann Arbor, United States	25 May 1935 ^[1]
8.21 m (26 ft 11¼ in)	0.0	 Ralph Boston (USA)	Walnut, United States	12 August 1960 ^[1]
8.24 m (27 ft ½ in)	1.8	 Ralph Boston (USA)	Modesto, United States	27 May 1961 ^[1]
8.28 m (27 ft 2 in)	1.2	 Ralph Boston (USA)	Moscow, Soviet Union	16 July 1961 ^[1]
8.31 m (27 ft 3¼ in)	-0.1	 Igor Ter-Ovanesyan (URS)	Yerevan, Soviet Union	10 June 1962 ^[1]
8.31 m (27 ft 3¼ in)	0.0	 Ralph Boston (USA)	Kingston, Jamaica	15 August 1964 ^[1]
8.34 m (27 ft 4¼ in)	1.0	 Ralph Boston (USA)	Los Angeles, United States	12 September 1964 ^[1]
8.35 m (27 ft 4¾ in)	0.0	 Ralph Boston (USA)	Modesto, United States	29 May 1965 ^[1]
8.35 m (27 ft 4¾ in) at Altitude	0.0	 Igor Ter-Ovanesyan (URS)	Mexico City, Mexico	19 October 1967 ^[1]
8.90 m (29 ft 2½ in) at Altitude	2.0	 Bob Beamon (USA)	Mexico City, Mexico	18 October 1968 ^[1]
8.95 m (29 ft 4¼ in)	0.3	 Mike Powell (USA)	Shinjuku, Tokyo, Japan	30 August 1991 ^[1]

Figure 19.1: Men's Long Jump World Record Progression

19.1 Import Data

To import the data of the Record Progression table you can use a couple of functions from the package `rvest`.

```
library(rvest)
```

```
wiki_jump <- 'https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression'

long_jump <- read_html(wiki_jump)
tbl <- html_table(html_node(long_jump, 'table'))
```

The function `read_html()` reads the html file of the wikipedia page. This will produce an object of type `"xml_document"` which we can further manipulate with other functions in `"rvest"`.

Because the *Record progression* data is in an html `table` node, you can use `html_node()` to locate such table in the XML document. And then *extract* it with `html_table()`.

```
str(tbl, vec.len = 1)
```

```
tibble [18 x 5] (S3: tbl_df/tbl/data.frame)
 $ Mark   : chr [1:18] "7.61 m (24 ft 11 ½ in)" ...
 $ Wind   : num [1:18] NA NA ...
 $ Athlete: chr [1:18] "Peter O'Connor (IRE)" ...
 $ Venue  : chr [1:18] "Dublin, Ireland" ...
 $ Date   : chr [1:18] "5 August 1901[1]" ...
```

As you can tell, the extracted table `tbl` is a data frame with 18 rows and 5 columns.

19.2 Extracting Meters

The first task consists of looking at the values in column `Mark`, and find how to retrieve the distance values expressed in meters. For example, the first element in `Mark` is:

```
tbl$Mark[1]
```

```
[1] "7.61 m (24 ft 11 ½ in)"
```

The goal is to obtain the number 7.61. One way to achieve this task is via the `substr()` function.

```
substr(tbl$Mark[1], start = 1, stop = 4)
```

```
[1] "7.61"
```

We can do that for the entire vector:

```
meters <- substr(tbl$Mark, start = 1, stop = 4)
meters
```

```
[1] "7.61" "7.69" "7.76" "7.89" "7.90" "7.93" "7.98" "8.13" "8.21" "8.24"
[11] "8.28" "8.31" "8.31" "8.34" "8.35" "8.35" "8.90" "8.95"
```

Notice that the meter values are not really numeric but character. In order to have `meters` as numbers, we should coerce them with `as.numeric()`

```
meters <- as.numeric(substr(tbl$Mark, start = 1, stop = 4))
meters
```

```
[1] 7.61 7.69 7.76 7.89 7.90 7.93 7.98 8.13 8.21 8.24 8.28 8.31 8.31 8.34 8.35
[16] 8.35 8.90 8.95
```

Extracting Meters with Regular Expressions

Instead of using the function `substr()` to obtain the distance values, let's see how to achieve the same task using regular expressions. To do this we must determine a pattern to be matched. So, what is the pattern that all the distance values (in meters) have in common?

```
mark1 <- tbl$Mark[1]
mark1
```

```
[1] "7.61 m (24 ft 11 ½ in)"
```

If you look at the `Mark` content, you will notice that the target pattern is formed by: a digit, followed by a dot, followed by two digits. Such pattern can be codified as: `"[0-9]\\.[0-9][0-9]"`. So let's test it and see if there's match:

```
str_detect(mark1, pattern = "[0-9]\\.[0-9][0-9]")
```

```
[1] TRUE
```

To extract the distance pattern we use `str_extract()`

```
str_extract(mark1, pattern = "[0-9]\\.[0-9][0-9]")
```

```
[1] "7.61"
```

And then we can apply it on the entire column to get:

```
str_extract(tbl$Mark, pattern = "[0-9]\\.[0-9][0-9]")
```

```
[1] "7.61" "7.69" "7.76" "7.89" "7.90" "7.93" "7.98" "8.13" "8.21" "8.24"  
[11] "8.28" "8.31" "8.31" "8.34" "8.35" "8.35" "8.90" "8.95"
```

19.3 Extracting Country

Consider the column `Athlete`. The first value corresponds to `Petter O'Connor` from Ireland.

```
tbl$Athlete[1]
```

```
[1] "Peter O'Connor (IRE)"
```

Let's create a vector `peter` for this athlete:

```
peter <- tbl$Athlete[1]
```

How can we get the country abbreviation?

```
substr(peter, nchar(peter)-4, nchar(peter))
```

```
[1] "(IRE)"
```

That works but it is preferable to exclude the parentheses, that is, the third to last character, as well as the last character:

```
substr(peter, nchar(peter)-3, nchar(peter)-1)
```



```
[1] "IRE"
```

Now we can apply the `substr()` command with all the athletes:

```
# extract country
substr(tbl$Athlete, nchar(tbl$Athlete)-4, nchar(tbl$Athlete))
```

```
[1] "(IRE)" "(USA)" "(USA)" "(USA)" "(USA)" "(HAI)" "(JPN)" "(USA)" "(USA)"
[10] "(USA)" "(USA)" "(URS)" "(USA)" "(USA)" "(USA)" "(URS)" "(USA)" "(USA)"
```

```
country <- substr(tbl$Athlete, nchar(tbl$Athlete)-3, nchar(tbl$Athlete)-1)
country
```

```
[1] "IRE" "USA" "USA" "USA" "USA" "HAI" "JPN" "USA" "USA" "USA" "USA" "URS"
[13] "USA" "USA" "USA" "URS" "USA" "USA"
```

19.4 Cleaning Dates

Now let's consider the values in column Date:

```
# first 5 dates
tbl$Date[1:5]
```

```
[1] "5 August 1901[1]" "23 July 1921[1]" "7 July 1924[1]" "13 June 1925[1]"
[5] "7 July 1928[1]"
```

Notice that all the date values are formed by the day-number, the name of the month, the year, and then the characters [1]. Obviously we don't need those last three characters [1].

```
date1 <- tbl$Date[1]
date1
```

```
[1] "5 August 1901[1]"
```

First let's see how to match the pattern [1]. Perhaps the first option that an inexperienced user would try is:

```
str_detect(date1, pattern = "[1]")
```

```
[1] TRUE
```

According to `str_detect()`, there's a match, so let's see what exactly "[1]" is matching:

```
str_match(date1, pattern = "[1]")
```

```
      [,1]  
[1,] "1"
```

Mmmm, not quite right. We are matching the character "1" but not "[1]". Why? Because brackets are metacharacters. So in order to match brackets *as brackets* we need to escape them:

```
str_match(date1, pattern = "\\[1\\]")
```

```
      [,1]  
[1,] "[1]"
```

Now we are talking. The next step involves using `str_replace()` to match the pattern "\\[1\\]" and replace it with an empty string "":

```
str_replace(date1, pattern = "\\[1\\]", replacement = "")
```

```
[1] "5 August 1901"
```

Then, we can get an entire vector of clean dates:

```
# clean dates  
dates <- str_replace(tbl$Date, pattern = "\\[1\\]", replacement = "")  
dates
```

```
[1] "5 August 1901"      "23 July 1921"      "7 July 1924"  
[4] "13 June 1925"       "7 July 1928"       "9 September 1928"  
[7] "27 October 1931"    "25 May 1935"       "12 August 1960"  
[10] "27 May 1961"        "16 July 1961"      "10 June 1962"  
[13] "15 August 1964"     "12 September 1964" "29 May 1965"  
[16] "19 October 1967"    "18 October 1968"   "30 August 1991"
```

19.5 Month and Day

We can further manipulate the dates. For example, say we are interested in extracting the name of the month. In the first date, this corresponds to extracting "August":

```
dates[1]
```

```
[1] "5 August 1901"
```

How can we do that? Several approaches can be applied in this case. For example, let's inspect the format of the month names:

```
dates[1:5]
```

```
[1] "5 August 1901" "23 July 1921" "7 July 1924" "13 June 1925"  
[5] "7 July 1928"
```

They all begin with an upper case letter, followed by the rest of the characters in lower case. If we want to match month names formed by four letters (e.g. June, July), we could look for the pattern "[A-Z] [a-z] [a-z] [a-z] "

```
str_extract(dates, pattern = "[A-Z] [a-z] [a-z] [a-z] ")
```

```
[1] "Augu" "July" "July" "June" "July" "Sept" "Octo" NA      "Augu" NA  
[11] "July" "June" "Augu" "Sept" NA      "Octo" "Octo" "Augu"
```

The previous pattern "[A-Z] [a-z] [a-z] [a-z] " not only matches "June" and "July" but also "Augu", "Sept", "Octo". In addition, we have some missing values.

Because the month names have variable lengths, we can use a repetition or quantifier operator. More specifically, we could look for the pattern "[A-Z] [a-z] +", that is: an upper case letter, followed by a lower case letter, repeated one or more times. The plus + tells the regex engine to attempt to match the preceding token once or more:

```
month_names <- str_extract(dates, pattern = "[A-Z] [a-z] +")  
month_names
```

```
[1] "August"      "July"        "July"        "June"        "July"        "September"
[7] "October"     "May"         "August"      "May"         "July"         "June"
[13] "August"     "September"  "May"         "October"     "October"     "August"
```

Having extracted the name of the months, we can take advantage of a similar pattern to extract the days. How? Using a pattern formed by one digit range and the plus sign: "[0-9]+"

```
str_extract(dates, pattern = "[0-9]+")
```

```
[1] "5"  "23" "7"  "13" "7"  "9"  "27" "25" "12" "27" "16" "10" "15" "12" "29"
[16] "19" "18" "30"
```

19.6 Extracting Year

What about extracting the year number?

```
dates[1]
```

```
[1] "5 August 1901"
```

One option that we have discussed already is to use `substr()` or `str_sub()`

```
str_sub(dates, start = nchar(dates)-3, end = nchar(dates))
```

```
[1] "1901" "1921" "1924" "1925" "1928" "1928" "1931" "1935" "1960" "1961"
[11] "1961" "1962" "1964" "1964" "1965" "1967" "1968" "1991"
```

or simply indicate a negative starting position (to counting from the end of the string):

```
str_sub(dates, start = -4)
```

```
[1] "1901" "1921" "1924" "1925" "1928" "1928" "1931" "1935" "1960" "1961"
[11] "1961" "1962" "1964" "1964" "1965" "1967" "1968" "1991"
```

Another option consists in using a pattern formed by four digits: "[0-9][0-9][0-9][0-9]":

```
str_extract(dates[1], pattern = "[0-9][0-9][0-9][0-9]")
```

```
[1] "1901"
```

An additional option consists in using an *end of string anchor* with the metacharacter "\$" (dollar sign), and combine with a repetition operator "+" like: "[0-9]+\$":

```
str_extract(dates[1], pattern = "[0-9]+$")
```

```
[1] "1901"
```

What is this pattern doing? The part of the pattern "[0-9]+" indicates that we want to match one or more digits. In order to tell the engine to match the pattern at the end of the string, we must use the anchor "\$".

The same task can be achieved with a digit character class "\\d" and the repetition operator "+":

```
str_extract(dates[1], pattern = "\\d+$")
```

```
[1] "1901"
```

19.7 Athlete Names

Now let's try to extract the athletes' first and last names. We could specify a regex pattern for the first name [A-Z][a-z][A-Z]?[a-z]+, followed by a space, followed by an upper case letter, and one or more lower case letters [A-Z][a-z]+:

```
str_extract(tbl$Athlete, pattern = "[A-Z][a-z][A-Z]?[a-z]+ [A-Z][a-z]+")
```

```
[1] NA "Edward Gourdin" "Robert Le" "DeHart Hubbard"
[5] "Edward Hamm" "Sylvio Cator" "Chuhei Nambu" "Jesse Owens"
[9] "Ralph Boston" "Ralph Boston" "Ralph Boston" "Igor Ter"
[13] "Ralph Boston" "Ralph Boston" "Ralph Boston" "Igor Ter"
[17] "Bob Beamon" "Mike Powell"
```

What about the first athlete Peter O'Connor? The previous pattern does not include the apostrophe.

```
# works for Peter O'Connor only
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z]' [A-Z] [a-z]+")
```

```
[1] "Peter O'Connor" NA NA NA
[5] NA NA NA NA
[9] NA NA NA NA
[13] NA NA NA NA
[17] NA NA NA NA
```

What about this other pattern?

```
# still only works for Peter O'Connor
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z]' [A-Z]?[a-z]+")
```

```
[1] "Peter O'Connor" NA NA NA
[5] NA NA NA NA
[9] NA NA NA NA
[13] NA NA NA NA
[17] NA NA NA NA
```

Recall that the quantifier (or repetition) operators have an effect on the preceding token. So, the pattern "[A-Z]' [A-Z]?[a-z]+" means: an upper case letter, followed by an apostrophe, followed by an optional upper case, followed by one or more lower case letters. In other words, the quantifier "?" only has an effect on the second upper case letter.

In reality, we want both the apostrophe and the second upper case letters to be optional, so we need to add quantifiers "?" to both of them:

```
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z]'?[A-Z]?[a-z]+")
```

```
[1] "Peter O'Connor" "Edward Gourdin" "Robert Le" "DeHart Hubbard"
[5] "Edward Hamm" "Sylvio Cator" "Chuhei Nambu" "Jesse Owens"
[9] "Ralph Boston" "Ralph Boston" "Ralph Boston" "Igor Ter"
[13] "Ralph Boston" "Ralph Boston" "Ralph Boston" "Igor Ter"
[17] "Bob Beamon" "Mike Powell"
```

If you want to treat a set of characters as a single unit, you must wrap them inside parentheses:

```
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z] (' [A-Z])?[a-z]+")
```

```
[1] "Peter O'Connor" "Edward Gourdin" "Robert Le"      "DeHart Hubbard"
[5] "Edward Hamm"    "Sylvio Cator"   "Chuhei Nambu"   "Jesse Owens"
[9] "Ralph Boston"   "Ralph Boston"   "Ralph Boston"   "Igor Ter"
[13] "Ralph Boston"   "Ralph Boston"   "Ralph Boston"   "Igor Ter"
[17] "Bob Beamon"     "Mike Powell"
```

We still have an issue with athlete Igor Ter-Ovanesyan. The patterns used so far are only matching the characters in his last name before the hyphen. We can start by adding an escaped hyphen inside the character set "[a-z\\-]" at the end of the pattern:

```
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z] (' [A-Z])?[a-z\\-]+")
```

```
[1] "Peter O'Connor" "Edward Gourdin" "Robert Le"      "DeHart Hubbard"
[5] "Edward Hamm"    "Sylvio Cator"   "Chuhei Nambu"   "Jesse Owens"
[9] "Ralph Boston"   "Ralph Boston"   "Ralph Boston"   "Igor Ter-"
[13] "Ralph Boston"   "Ralph Boston"   "Ralph Boston"   "Igor Ter-"
[17] "Bob Beamon"     "Mike Powell"
```

Notice that this pattern does match the hyphen but fails to match the second part of the last name (the one after the hyphen). This is because our token is only matching lower case letters. So we also need to include upper case letters in the character set: "[a-zA-Z\\-]"

```
str_extract(tbl$Athlete, pattern = "[A-Z] [a-z] [A-Z]?[a-z]+ [A-Z] (' [A-Z])?[a-zA-Z\\-]+")
```

```
[1] "Peter O'Connor"      "Edward Gourdin"      "Robert LeGendre"
[4] "DeHart Hubbard"     "Edward Hamm"         "Sylvio Cator"
[7] "Chuhei Nambu"       "Jesse Owens"         "Ralph Boston"
[10] "Ralph Boston"       "Ralph Boston"        "Igor Ter-Ovanesyan"
[13] "Ralph Boston"       "Ralph Boston"        "Ralph Boston"
[16] "Igor Ter-Ovanesyan" "Bob Beamon"          "Mike Powell"
```

The regex patterns that involve a set such as "[a-zA-Z]" can be simplified with a repeated **word** character class "\\w+" (recall that "\\w+" is equivalent to "[0-9A-Za-z_]"). We can try to use two repeated word classes:

```
str_extract(tbl$Athlete, pattern = "\\w+ \\w+")
```

[1]	"Peter O"	"Edward Gourdin"	"Robert LeGendre"	"DeHart Hubbard"
[5]	"Edward Hamm"	"Sylvio Cator"	"Chuhei Nambu"	"Jesse Owens"
[9]	"Ralph Boston"	"Ralph Boston"	"Ralph Boston"	"Igor Ter"
[13]	"Ralph Boston"	"Ralph Boston"	"Ralph Boston"	"Igor Ter"
[17]	"Bob Beamon"	"Mike Powell"		

As you know, we also need to include an apostrophe and the hyphen. In this case, we can include them inside parentheses and separating them with the OR operator "|":

```
str_extract(tbl$Athlete, pattern = "\\w+ (\\w|-|')+")
```

[1]	"Peter O'Connor"	"Edward Gourdin"	"Robert LeGendre"
[4]	"DeHart Hubbard"	"Edward Hamm"	"Sylvio Cator"
[7]	"Chuhei Nambu"	"Jesse Owens"	"Ralph Boston"
[10]	"Ralph Boston"	"Ralph Boston"	"Igor Ter-Ovanesyan"
[13]	"Ralph Boston"	"Ralph Boston"	"Ralph Boston"
[16]	"Igor Ter-Ovanesyan"	"Bob Beamon"	"Mike Powell"

20 Data Log File

In this example, we'll be using the text file `logfile.txt` located in the `data/` folder of the book's github repository:

<https://raw.githubusercontent.com/gastonstat/strings-data/main/data/logfile.txt>

This file is a **server log file** that contains the recorded events taking place in a web server. The content of the file is in a special format known as *common log format*.

https://en.wikipedia.org/wiki/Common_Log_Format

According to wikipedia:

“The Common Log Format is a standardized text file format used by web servers when generating server log files.”

Here's an example of a log record; the text should be in one line of code, but I've split it into 2 lines for readability purposes:

```
pd9049dac.dip.t-dialin.net - - [01/May/2001:01:51:25 -0700]  
"GET /accesswatch/accesswatch-1.33/ HTTP/1.0" 200 1004
```

- A "-" in a field indicates missing data.
- `pd9049dac.dip.t-dialin.net` is the IP address of the client (remote host) which made the request to the server.
- `[01/May/2001:01:51:25 -0700]` is the date, time, and time zone that the request was received, by default in strftime format `%d/%b/%Y:%H:%M:%S %z`.
- `"GET /accesswatch/accesswatch-1.33/ HTTP/1.0"` is the request line from the client.
- The method `GET`, `/accesswatch/accesswatch-1.33/` is the resource requested, and `HTTP/1.0` is the HTTP protocol.
- 200 is the HTTP status code returned to the client.
 - 2xx is a successful response
 - 3xx a redirection
 - 4xx a client error, and
 - 5xx a server error
- 1004 is the size of the object returned to the client, measured in bytes.

If you want to download a copy of the text file to your working directory (from within R) run the following code:

```
# download file
github <- "https://raw.githubusercontent.com/gastonstat/strings-data"
textfile <- "/main/data/logfile.txt"
download.file(url = paste0(github, textfile), destfile = "logfile.txt")
```

20.1 Reading the text file

The first step involves reading the data in R. How can you do this? One option is with the `readLines()` function which reads any text file into a character vector:

```
# one option is to read in the content with 'readLines()'
logs <- readLines('logfile.txt')
```

Let's take a peek at the content of the vector `logs` by running `head(logs)`. We display the output of the first three lines below::

```
# take a peek at the contents in logs
head(logs)
```

```
[1] "pd9049dac.dip.t-dialin.net - - [01/May/2001:01:51:25 -0700] \"GET /accesswatch/accesswa"
[2] "pd9049dac.dip.t-dialin.net - - [01/May/2001:01:51:26 -0700] \"GET /accesswatch/accesswa"
[3] "pd9049dac.dip.t-dialin.net - - [01/May/2001:01:51:26 -0700] \"GET /sa.inside.jpg HTTP/1"
[4] "pd9049dac.dip.t-dialin.net - - [01/May/2001:02:20:19 -0700] \"GET /accesswatch/accesswa"
[5] "pd9049dac.dip.t-dialin.net - - [01/May/2001:02:20:20 -0700] \"GET /accesswatch/accesswa"
[6] "pd9049dac.dip.t-dialin.net - - [01/May/2001:02:20:20 -0700] \"GET /accesswatch/accesswa"
```

Because the file contains 26033 lines (or elements), let's get a subset by taking a random sample of size 50:

```
# subset a sample of lines
set.seed(98765)
s <- sample(1:length(logs), size = 50)
sublogs <- logs[s]
```

20.1.1 JPG File Requests

To begin our regex experiments, let's try to find out "how many requests involved a JPG file?". One way to answer the previous question is by counting the number of lines containing the pattern "jpg". We can use `grep()` to match or detect this pattern:

```
# matching "jpg" (which elements)
grep("jpg", sublogs)
```

```
[1]  2  6  8 10 11 14 26 34 35 36 40 41 45
```

```
# showing value of matches
grep("jpg", sublogs, value = TRUE)
```

```
[1] "hj.a11.betware.com - - [29/May/2001:02:15:26 -0700] \"GET /testing/images/archi_servle
[2] "port194.ds1-gl.adsl.cybercity.dk - - [29/May/2001:15:33:04 -0700] \"GET /testing/images
[3] "pd9049e9b.dip.t-dialin.net - - [06/May/2001:12:45:35 -0700] \"GET /accesswatch/accessw
[4] "pd9049e94.dip.t-dialin.net - - [04/May/2001:04:12:53 -0700] \"GET /accesswatch/accessw
[5] "edslink9.eds.com - - [23/May/2001:02:22:12 -0700] \"GET /testing/images/pshtk.jpg HTTP
[6] "bloodymary.vebis.de - - [22/May/2001:01:15:28 -0700] \"GET /testing/images/scope.jpg H
[7] "line210-137.iplan.com.ar - - [28/May/2001:14:49:26 -0700] \"GET /testing/images/archi_
[8] "aannecy-101-2-1-40.abo.wanadoo.fr - - [31/May/2001:08:43:33 -0700] \"GET /testing/imag
[9] "rr-2s01.inf.fh-rhein-sieg.de - - [23/May/2001:06:40:00 -0700] \"GET /xp-r/img17.jpg HT
[10] "pd9541d23.dip.t-dialin.net - - [29/May/2001:07:08:26 -0700] \"GET /sa.inside.jpg HTTP/
[11] "rose.ap.dregis.com - - [29/May/2001:05:09:28 -0700] \"GET /testing/images/archi.jpg HT
[12] "208.36.196.8 - - [23/May/2001:22:39:40 -0700] \"GET /testing/images/archi.jpg HTTP/1.1
[13] "hlch0enk.htc.com - - [22/May/2001:11:01:53 -0700] \"GET /testing/images/archi.jpg HTTP/
```

We can try to be more specific by defining a pattern ".jpg" in which the . corresponds to the *literal* dot character. To match the dot, we need to escape it with "\\.":

```
# we could try to be more precise and match ".jpg"
grep("\\.jpg ", sublogs, value = TRUE)
```

```
[1] "hj.a11.betware.com - - [29/May/2001:02:15:26 -0700] \"GET /testing/images/archi_servle
[2] "port194.ds1-gl.adsl.cybercity.dk - - [29/May/2001:15:33:04 -0700] \"GET /testing/images
[3] "pd9049e9b.dip.t-dialin.net - - [06/May/2001:12:45:35 -0700] \"GET /accesswatch/accessw
[4] "pd9049e94.dip.t-dialin.net - - [04/May/2001:04:12:53 -0700] \"GET /accesswatch/accessw
```

```
[5] "edslink9.eds.com - - [23/May/2001:02:22:12 -0700] \"GET /testing/images/pshtk.jpg HTTP/1.1\" 200 1024
[6] "bloodymary.vebis.de - - [22/May/2001:01:15:28 -0700] \"GET /testing/images/scope.jpg HTTP/1.1\" 200 1024
[7] "line210-137.iplan.com.ar - - [28/May/2001:14:49:26 -0700] \"GET /testing/images/archi.jpg HTTP/1.1\" 200 1024
[8] "aannecy-101-2-1-40.abo.wanadoo.fr - - [31/May/2001:08:43:33 -0700] \"GET /testing/images/archi.jpg HTTP/1.1\" 200 1024
[9] "rr-2s01.inf.fh-rhein-sieg.de - - [23/May/2001:06:40:00 -0700] \"GET /xp-r/img17.jpg HTTP/1.1\" 200 1024
[10] "pd9541d23.dip.t-dialin.net - - [29/May/2001:07:08:26 -0700] \"GET /sa.inside.jpg HTTP/1.1\" 200 1024
[11] "rose.ap.dregis.com - - [29/May/2001:05:09:28 -0700] \"GET /testing/images/archi.jpg HTTP/1.1\" 200 1024
[12] "208.36.196.8 - - [23/May/2001:22:39:40 -0700] \"GET /testing/images/archi.jpg HTTP/1.1\" 200 1024
[13] "hlch0enk.htc.com - - [22/May/2001:11:01:53 -0700] \"GET /testing/images/archi.jpg HTTP/1.1\" 200 1024
```

A similar output of `grep()` can be obtained with `str_detect()`, which allows you to *detect* what elements contain a match to the specified pattern:

```
# matching "jpg" (which elements)
str_detect(string = sublogs, pattern = "\\.(jpg)")
```

```
[1] FALSE TRUE FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE TRUE FALSE
[13] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
[37] FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
[49] FALSE FALSE
```

We can do the same for PNG extensions (or for GIF or ICO):

```
# matching "png" (which elements)
str_detect(string = sublogs, pattern = "\\.(png)")
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[49] FALSE TRUE
```

20.1.2 Extracting file extensions

Another common task when working with regular expressions has to do with pattern extraction. For this purposes, we can use `str_extract()`:

```
# extracting "jpg" (which lements)
str_extract(string = sublogs, pattern = "\\ .jpg")
```

```
[1] NA      ".jpg" NA      NA      NA      ".jpg" NA      ".jpg" NA      ".jpg"
[11] ".jpg" NA      NA      ".jpg" NA      NA      NA      NA      NA      NA
[21] NA      NA      NA      NA      NA      ".jpg" NA      NA      NA      NA
[31] NA      NA      NA      ".jpg" ".jpg" ".jpg" NA      NA      NA      ".jpg"
[41] ".jpg" NA      NA      NA      ".jpg" NA      NA      NA      NA      NA
```

`str_extract()` actually let us confirm that we are matching the desired patterns. Notice that when there is no match, `str_extract()` returns a missing value NA.

20.1.3 Image files

Now let's try to detect all types of image files: JPG, PNG, GIF, ICO

```
# looking for image file extensions
jpgs <- str_detect(logs, pattern = "\\ .jpg ")
sum(jpgs)
```

```
[1] 5509
```

```
pngs <- str_detect(logs, pattern = "\\ .png ")
sum(pngs)
```

```
[1] 1374
```

```
gifs <- str_detect(logs, pattern = "\\ .gif")
sum(gifs)
```

```
[1] 8818
```

```
icos <- str_detect(logs, pattern = "\\ .ico ")
sum(icos)
```

```
[1] 100
```

20.1.4 How to match image files with one regex pattern?

We can use character sets to define a more generic pattern. For instance, to match "jpg" or "png", we could join three character sets: "[jp][pn][g]". The first set [jp] looks for j or p, the second set [pn] looks for p or n, and the third set simply looks for g.

```
# matching "jpg" or "png"
jpg_png_lines <- str_detect(sublogs, "[jp][pn][g]")
sum(jpg_png_lines)
```

```
[1] 15
```

Including the dot, we can use: "\\.[jp][pn][g]"

```
# matching "jpg" or "png"
jpg_png_lines <- str_detect(sublogs, "\\.[jp][pn][g]")
sum(jpg_png_lines)
```

```
[1] 15
```

We could generalize the pattern to include the GIF and ICO extensions:

```
# matching "jpg" or "png" or "gif"
image_lines1 <- str_detect(sublogs, "[jpgi][pnig][gfo]")
sum(image_lines1)
```

```
[1] 44
```

To confirm that we are actually matching jpg, png, gif and ico, let's use `str_extract()`

```
# are we correctly extracting image file extensions?
str_extract(sublogs, "[jpgi][pnig][gfo]")
```

```
[1] "ing" "ing" NA    "ing" "ing" "ing" "ing" "jpg" "ing" "jpg" "ing" "ing"
[13] NA    "ing" "ing" "ing" "ing" "ing" "ing" "gif" "ing" "ico" "ing" "ing"
[25] "ing" "ing" "ing" NA    "pco" "ing" "gif" "ing" NA    "ing" "inf" "jpg"
[37] "gif" "ing" "ing" "ing" "ing" "gif" "ing" "ing" "ing" "ing" NA    NA
[49] "gif" "ing"
```

The previous pattern does not really work as expected: note that we are matching the patterns formed by "ing" and "inf" which do not correspond to image file extensions.

An alternative way to detect JPG and PNG is by grouping patterns inside parentheses, and separating them with the metacharacter "|" which means *OR*:

```
# detecting .jpg OR .png
jpg_png <- str_detect(sublogs, "\\.(jpg|png)")
sum(jpg_png)
```

```
[1] 15
```

Here's how to detect all the extension in one single pattern:

```
# matching "jpg" or "png" or "gif" or "ico"
image_lines <- str_detect(sublogs, "\\.(jpg|png|gif|ico)")
sum(image_lines)
```

```
[1] 28
```

To make sure our regex operation is successful, let's see the output of `str_extract()`:

```
images_output <- str_extract(sublogs, "\\.(jpg|png|gif|ico)")
images_output
```

```
[1] NA      ".jpg" NA      NA      NA      ".jpg" ".gif" ".jpg" NA      ".jpg"
[11] ".jpg" NA      NA      ".jpg" NA      ".png" NA      NA      NA      ".gif"
[21] ".gif" ".gif" NA      NA      ".gif" ".jpg" NA      NA      ".gif" NA
[31] ".gif" ".gif" NA      ".jpg" ".jpg" ".jpg" ".gif" NA      NA      ".jpg"
[41] ".jpg" ".gif" NA      ".gif" ".jpg" ".gif" NA      NA      ".gif" ".png"
```

There's some repetition with the dot character; we can modify our previous pattern by placing the dot "\\." at the beginning:

```
images_output <- str_extract(sublogs, "\\.(jpg|png|gif|ico)")
images_output
```

```

[1] NA      ".jpg" NA      NA      NA      ".jpg" "gif"  ".jpg" NA      ".jpg"
[11] ".jpg" NA      NA      ".jpg" NA      "png"  NA      NA      NA      "gif"
[21] "gif"  "ico"  NA      NA      "gif"  ".jpg" NA      NA      "gif"  NA
[31] "gif"  "gif"  NA      ".jpg" ".jpg" ".jpg" "gif"  NA      NA      ".jpg"
[41] ".jpg" "gif"  NA      "gif"  ".jpg" "gif"  NA      NA      "gif"  "png"

```

Notice that the dot only appears next to ".jpg" but not with the other type of extensions. What we need to do is group the file extensions by surrounding them with parentheses:

```

images_output <- str_extract(sublogs, "\\.(jpg|png|gif|ico)")
images_output

```

```

[1] NA      ".jpg" NA      NA      NA      ".jpg" ".gif" ".jpg" NA      ".jpg"
[11] ".jpg" NA      NA      ".jpg" NA      ".png" NA      NA      NA      ".gif"
[21] ".gif" ".gif" NA      NA      ".gif" ".jpg" NA      NA      ".gif" NA
[31] ".gif" ".gif" NA      ".jpg" ".jpg" ".jpg" ".gif" NA      NA      ".jpg"
[41] ".jpg" ".gif" NA      ".gif" ".jpg" ".gif" NA      NA      ".gif" ".png"

```

Now let's apply the pattern on the entire log file, to count the number of files of each type:

```

# frequencies
img_extensions <- str_extract(logs, "\\.(jpg|png|gif|ico)")
table(img_extensions)

```

```

img_extensions
.gif .ico .jpg .png
8818 100 5509 1374

```