**Manipulate strings with {stringr}**

{stringr} contains functions to manipulate strings.

I will discuss the most common string operations: detecting, locating, matching, searching and  
replacing, and exctracting/removing strings.

The file looks like this:

timole

tlnldre

timor

insole

landed

verc

veer

tll

Cu

tall

We are interested in the strings after CONTENT=. We are going to use functions from the {stringr}  
package to get the strings after CONTENT=. In Chapter 10, we are going to explore this file  
again, but using complex regular expressions to get all the content in one go.

**Getting text data into Rstudio**

First of all, let us read in the file:

winchester <- read\_lines("https://gist.githubusercontent.com/b-rodrigues/5139560e7d0f2ecebe5da1df3629e015/raw/e3031d894ffb97217ddbad1ade1b307c9937d2c8/gistfile1.txt")

Even though the file is an XML file, I still read it in using read\_lines() and not read\_xml()  
from the {xml2} package. This is for the purposes of the current exercise, and also because I  
always have trouble with XML files, and prefer to treat them as simple text files, and use regular  
expressions to get what I need.

Now that the ALTO file is read in and saved in the winchester variable, you might want to print  
the whole thing in the console. Before that, take a look at the structure:

str(winchester)

## chr [1:43] "" ...

So the winchester variable is a character atomic vector with 43 elements. So first, we need to  
understand what these elements are. Let’s start with the first one:

winchester[1]

## [1] ""

Ok, so it seems like the first element is part of the header of the file. What about the second one?

winchester[2]

## [1] "

This is Google's cache of https://chroniclingamerica.loc.gov/lccn/sn86069133/1910-10-31/ed-1/seq-1/ocr.xml. It is a snapshot of the page as it appeared on 21 Jan 2019 05:18:18 GMT. The current page could have changed in the meantime. Learn more.

Full versionText-only versionView source

Tip: To quickly find your search term on this page, press **Ctrl+F** or **⌘-F** (Mac) and use the find bar.

"

Same. So where is the content? The file is very large, so if you print it in the console, it will  
take quite some time to print, and you will not really be able to make out anything. The best  
way would be to try to detect the string CONTENT and work from there.

**Detecting, getting the position and locating strings**

When confronted to an atomic vector of strings, you might want to know inside which elements you  
can find certain strings. For example, to know which elements of winchester contain the string  
CONTENT, use str\_detect():

winchester %>%

str\_detect("CONTENT")

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

## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

This returns a boolean atomic vector of the same length as winchester. If the string CONTENT is  
nowhere to be found, the result will equal FALSE, if not it will equal TRUE. Here it is easy to  
see that the last element contains the string CONTENT. But what if instead of having 43 elements,  
the vector had 24192 elements? And hundreds would contain the string CONTENT? It would be easier  
to instead have the indices of the vector where one can find the word CONTENT. This is possible  
with str\_which():

winchester %>%

str\_which("CONTENT")

## [1] 43

Here, the result is 43, meaning that the 43rd element of winchester contains the string CONTENT  
somewhere. If we need more precision, we can use str\_locate() and str\_locate\_all(). To explain  
how both these functions work, let’s create a very small example:

ancient\_philosophers <- c("aristotle", "plato", "epictetus", "seneca the younger", "epicurus", "marcus aurelius")

Now suppose I am interested in philosophers whose name ends in us. Let us use str\_locate() first:

ancient\_philosophers %>%

str\_locate("us")

## start end

## [1,] NA NA

## [2,] NA NA

## [3,] 8 9

## [4,] NA NA

## [5,] 7 8

## [6,] 5 6

You can interpret the result as follows: in the rows, the index of the vector where the  
string us is found. So the 3rd, 5th and 6th philosopher have us somewhere in their name.  
The result also has two columns: start and end. These give the position of the string. So the  
string us can be found starting at position 8 of the 3rd element of the vector, and ends at position  
9. Same goes for the other philisophers. However, consider Marcus Aurelius. He has two names, both  
ending with us. However, str\_locate() only shows the position of the us in Marcus.

To get both us strings, you need to use str\_locate\_all():

ancient\_philosophers %>%

str\_locate\_all("us")

## [[1]]

## start end

##

## [[2]]

## start end

##

## [[3]]

## start end

## [1,] 8 9

##

## [[4]]

## start end

##

## [[5]]

## start end

## [1,] 7 8

##

## [[6]]

## start end

## [1,] 5 6

## [2,] 14 15

Now we get the position of the two us in Marcus Aurelius. Doing this on the winchester vector  
will give use the position of the CONTENT string, but this is not really important right now. What  
matters is that you know how str\_locate() and str\_locate\_all() work.

So now that we know what interests us in the 43nd element of winchester, let’s take a closer  
look at it:

winchester[43]

As you can see, it’s a mess:

PrcidehtPridesuccesssoarcencent

The file was imported without any newlines. So we need to insert them ourselves, by splitting the  
string in a clever way.

**Splitting strings**

There are two functions included in {stringr} to split strings, str\_split() and str\_split\_fixed().  
Let’s go back to our ancient philosophers. Two of them, Seneca the Younger and Marcus Aurelius have  
something else in common than both being Roman Stoic philosophers. Their names are composed of several  
words. If we want to split their names at the space character, we can use str\_split() like this:

ancient\_philosophers %>%

str\_split(" ")

## [[1]]

## [1] "aristotle"

##

## [[2]]

## [1] "plato"

##

## [[3]]

## [1] "epictetus"

##

## [[4]]

## [1] "seneca" "the" "younger"

##

## [[5]]

## [1] "epicurus"

##

## [[6]]

## [1] "marcus" "aurelius"

str\_split() also has a simplify = TRUE option:

ancient\_philosophers %>%

str\_split(" ", simplify = TRUE)

## [,1] [,2] [,3]

## [1,] "aristotle" "" ""

## [2,] "plato" "" ""

## [3,] "epictetus" "" ""

## [4,] "seneca" "the" "younger"

## [5,] "epicurus" "" ""

## [6,] "marcus" "aurelius" ""

This time, the returned object is a matrix.

What about str\_split\_fixed()? The difference is that here you can specify the number of pieces  
to return. For example, you could consider the name “Aurelius” to be the middle name of Marcus Aurelius,  
and the “the younger” to be the middle name of Seneca the younger. This means that you would want  
to split the name only at the first space character, and not at all of them. This is easily achieved  
with str\_split\_fixed():

ancient\_philosophers %>%

str\_split\_fixed(" ", 2)

## [,1] [,2]

## [1,] "aristotle" ""

## [2,] "plato" ""

## [3,] "epictetus" ""

## [4,] "seneca" "the younger"

## [5,] "epicurus" ""

## [6,] "marcus" "aurelius"

This gives the expected result.

So how does this help in our case? Well, if you look at how the ALTO file looks like, at the beginning  
of this section, you will notice that every line ends with the “>” character. So let’s split at  
that character!

winchester\_text <- winchester[43] %>%

str\_split(">")

Let’s take a closer look at winchester\_text:

str(winchester\_text)

## List of 1

## $ : chr [1:19706] "

So this is a list of length one, and the first, and only, element of that list is an atomic vector  
with 19706 elements. Since this is a list of only one element, we can simplify it by saving the  
atomic vector in a variable:

winchester\_text <- winchester\_text[[1]]

Let’s now look at some lines:

winchester\_text[1232:1245]

## [1] "

This now looks easier to handle. We can narrow it down to the lines that only contain the string  
we are interested in, “CONTENT”. First, let’s get the indices:

content\_winchester\_index <- winchester\_text %>%

str\_which("CONTENT")

How many lines contain the string “CONTENT”?

length(content\_winchester\_index)

## [1] 4462

As you can see, this reduces the amount of data we have to work with. Let us save this is a new  
variable:

content\_winchester <- winchester\_text[content\_winchester\_index]

**Matching strings**

Matching strings is useful, but only in combination with regular expressions. As stated at the  
beginning of this section, we are going to learn about regular expressions in Chapter 10, but in  
order to make this section useful, we are going to learn the easiest, but perhaps the most useful  
regular expression: .\*.

Let’s go back to our ancient philosophers, and use str\_match() and see what happens. Let’s match  
the “us” string:

ancient\_philosophers %>%

str\_match("us")

## [,1]

## [1,] NA

## [2,] NA

## [3,] "us"

## [4,] NA

## [5,] "us"

## [6,] "us"

Not very useful, but what about the regular expression .\*? How could it help?

ancient\_philosophers %>%

str\_match(".\*us")

## [,1]

## [1,] NA

## [2,] NA

## [3,] "epictetus"

## [4,] NA

## [5,] "epicurus"

## [6,] "marcus aurelius"

That’s already very interesting! So how does .\* work? To understand, let’s first start by using  
. alone:

ancient\_philosophers %>%

str\_match(".us")

## [,1]

## [1,] NA

## [2,] NA

## [3,] "tus"

## [4,] NA

## [5,] "rus"

## [6,] "cus"

This also matched whatever symbol comes just before the “u” from “us”. What if we use two . instead?

ancient\_philosophers %>%

str\_match("..us")

## [,1]

## [1,] NA

## [2,] NA

## [3,] "etus"

## [4,] NA

## [5,] "urus"

## [6,] "rcus"

This time, we get the two symbols that immediately precede “us”. Instead of continuing like this  
we now use the \*, which matches zero or more of .. So by combining \* and ., we can match  
any symbol repeatedly, until there is nothing more to match. Note that there is also +, which works  
similarly to \*, but it matches one or more symbols.

There is also a str\_match\_all():

ancient\_philosophers %>%

str\_match\_all(".\*us")

## [[1]]

## [,1]

##

## [[2]]

## [,1]

##

## [[3]]

## [,1]

## [1,] "epictetus"

##

## [[4]]

## [,1]

##

## [[5]]

## [,1]

## [1,] "epicurus"

##

## [[6]]

## [,1]

## [1,] "marcus aurelius"

In this particular case it does not change the end result, but keep it in mind for cases like this one:

c("haha", "huhu") %>%

str\_match("ha")

## [,1]

## [1,] "ha"

## [2,] NA

and:

c("haha", "huhu") %>%

str\_match\_all("ha")

## [[1]]

## [,1]

## [1,] "ha"

## [2,] "ha"

##

## [[2]]

## [,1]

What if we want to match names containing the letter “t”? Easy:

ancient\_philosophers %>%

str\_match(".\*t.\*")

## [,1]

## [1,] "aristotle"

## [2,] "plato"

## [3,] "epictetus"

## [4,] "seneca the younger"

## [5,] NA

## [6,] NA

So how does this help us with our historical newspaper? Let’s try to get the strings that come  
after “CONTENT”:

winchester\_content <- winchester\_text %>%

str\_match("CONTENT.\*")

Let’s use our faithful str() function to take a look:

winchester\_content %>%

str

## chr [1:19706, 1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA ...

Hum, there’s a lot of NA values! This is because a lot of the lines from the file did not have the  
string “CONTENT”, so there is no match possible. Let’s us remove all these NAs. Because the  
result is a matrix, we cannot use the filter() function from {dplyr}. So we need to convert it  
to a tibble first:

winchester\_content <- winchester\_content %>%

as.tibble() %>%

filter(!is.na(V1))

## Warning: `as.tibble()` is deprecated, use `as\_tibble()` (but mind the new semantics).

## This warning is displayed once per session.

Because matrix columns do not have names, when a matrix gets converted into a tibble, the firt column  
gets automatically called V1. This is why I filter on this column. Let’s take a look at the data:

head(winchester\_content)

## # A tibble: 6 x 1

## V1

##

## 1 "CONTENT=\"J\" WC=\"0.8095238\"/"

## 2 "CONTENT=\"a\" WC=\"0.8095238\"/"

## 3 "CONTENT=\"Ira\" WC=\"0.95238096\"/"

## 4 "CONTENT=\"mj\" WC=\"0.8095238\"/"

## 5 "CONTENT=\"iI\" WC=\"0.8095238\"/"

## 6 "CONTENT=\"tE1r\" WC=\"0.8095238\"/"

**Searching and replacing strings**

We are getting close to the final result. We still need to do some cleaning however. Since our data  
is inside a nice tibble, we might as well stick with it. So let’s first rename the column and  
change all the strings to lowercase:

winchester\_content <- winchester\_content %>%

mutate(content = tolower(V1)) %>%

select(-V1)

Let’s take a look at the result:

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 "content=\"j\" wc=\"0.8095238\"/"

## 2 "content=\"a\" wc=\"0.8095238\"/"

## 3 "content=\"ira\" wc=\"0.95238096\"/"

## 4 "content=\"mj\" wc=\"0.8095238\"/"

## 5 "content=\"ii\" wc=\"0.8095238\"/"

## 6 "content=\"te1r\" wc=\"0.8095238\"/"

The second part of the string, “wc=….” is not really interesting. Let’s search and replace this  
with an empty string, using str\_replace():

winchester\_content <- winchester\_content %>%

mutate(content = str\_replace(content, "wc.\*", ""))

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 "content=\"j\" "

## 2 "content=\"a\" "

## 3 "content=\"ira\" "

## 4 "content=\"mj\" "

## 5 "content=\"ii\" "

## 6 "content=\"te1r\" "

We need to use the regular expression from before to replace “wc” and every character that follows.  
The same can be use to remove “content=”:

winchester\_content <- winchester\_content %>%

mutate(content = str\_replace(content, "content=", ""))

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 "\"j\" "

## 2 "\"a\" "

## 3 "\"ira\" "

## 4 "\"mj\" "

## 5 "\"ii\" "

## 6 "\"te1r\" "

We are almost done, but some cleaning is still necessary:

**Exctracting or removing strings**

Now, because I now the ALTO spec, I know how to find words that are split between two sentences:

winchester\_content %>%

filter(str\_detect(content, "hyppart"))

## # A tibble: 64 x 1

## content

##

## 1 "\"aver\" subs\_type=\"hyppart1\" subs\_content=\"average\" "

## 2 "\"age\" subs\_type=\"hyppart2\" subs\_content=\"average\" "

## 3 "\"considera\" subs\_type=\"hyppart1\" subs\_content=\"consideration\" "

## 4 "\"tion\" subs\_type=\"hyppart2\" subs\_content=\"consideration\" "

## 5 "\"re\" subs\_type=\"hyppart1\" subs\_content=\"resigned\" "

## 6 "\"signed\" subs\_type=\"hyppart2\" subs\_content=\"resigned\" "

## 7 "\"install\" subs\_type=\"hyppart1\" subs\_content=\"installed\" "

## 8 "\"ed\" subs\_type=\"hyppart2\" subs\_content=\"installed\" "

## 9 "\"be\" subs\_type=\"hyppart1\" subs\_content=\"before\" "

## 10 "\"fore\" subs\_type=\"hyppart2\" subs\_content=\"before\" "

## # … with 54 more rows

For instance, the word “average” was split over two lines, the first part of the word, “aver” on the  
first line, and the second part of the word, “age”, on the second line. We want to keep what comes  
after “subs\_content”. Let’s extract the word “average” using str\_extract(). However, because only  
some words were split between two lines, we first need to detect where the string “hyppart1” is  
located, and only then can we extract what comes after “subs\_content”. Thus, we need to combine  
str\_detect() to first detect the string, and then str\_extract() to extract what comes after  
“subs\_content”:

winchester\_content <- winchester\_content %>%

mutate(content = if\_else(str\_detect(content, "hyppart1"),

str\_extract\_all(content, "content=.\*", simplify = TRUE),

content))

Let’s take a look at the result:

winchester\_content %>%

filter(str\_detect(content, "content"))

## # A tibble: 64 x 1

## content

##

## 1 "content=\"average\" "

## 2 "\"age\" subs\_type=\"hyppart2\" subs\_content=\"average\" "

## 3 "content=\"consideration\" "

## 4 "\"tion\" subs\_type=\"hyppart2\" subs\_content=\"consideration\" "

## 5 "content=\"resigned\" "

## 6 "\"signed\" subs\_type=\"hyppart2\" subs\_content=\"resigned\" "

## 7 "content=\"installed\" "

## 8 "\"ed\" subs\_type=\"hyppart2\" subs\_content=\"installed\" "

## 9 "content=\"before\" "

## 10 "\"fore\" subs\_type=\"hyppart2\" subs\_content=\"before\" "

## # … with 54 more rows

We still need to get rid of the string “content=” and then of all the strings that contain “hyppart2”,  
which are not needed now:

winchester\_content <- winchester\_content %>%

mutate(content = str\_replace(content, "content=", "")) %>%

mutate(content = if\_else(str\_detect(content, "hyppart2"), NA\_character\_, content))

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 "\"j\" "

## 2 "\"a\" "

## 3 "\"ira\" "

## 4 "\"mj\" "

## 5 "\"ii\" "

## 6 "\"te1r\" "

Almost done! We only need to remove the " characters:

winchester\_content <- winchester\_content %>%

mutate(content = str\_replace\_all(content, "\"", ""))

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 "j "

## 2 "a "

## 3 "ira "

## 4 "mj "

## 5 "ii "

## 6 "te1r "

Let’s remove space characters with str\_trim():

winchester\_content <- winchester\_content %>%

mutate(content = str\_trim(content))

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 j

## 2 a

## 3 ira

## 4 mj

## 5 ii

## 6 te1r

To finish off this section, let’s remove stop words (words that do not add any meaning to a sentence,  
such as “as”, “and”…) and words that are composed of less than 3 characters. You can find a dataset  
with stopwords inside the {stopwords} package:

library(stopwords)

data(data\_stopwords\_stopwordsiso)

eng\_stopwords <- tibble("content" = data\_stopwords\_stopwordsiso$en)

winchester\_content <- winchester\_content %>%

anti\_join(eng\_stopwords) %>%

filter(nchar(content) > 3)

## Joining, by = "content"

head(winchester\_content)

## # A tibble: 6 x 1

## content

##

## 1 te1r

## 2 jilas

## 3 edition

## 4 winchester

## 5 news

## 6 injuries

That’s it for this section! You now know how to work with strings, but in Chapter 10 we are going  
one step further by learning about regular expressions, which offer much more power.