Regular Expressions with Stringr

Introduction to Data Science Workshop @ Hertie School

Anna Deniz & Ángela Duarte

So...no, this is not a cat on a keyboard, just Regexps

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
```

Copy

Introduction

Regular Expressions, also known as Regexps, are used to describe and find patterns in strings. They are useful not only in R but in other languages as well.

They may look ***very*** complicated if you are seeing them for the first time, but it is actually not that hard once you understand how they work.

In this tutorial, we are going to show you how to apply them using Stringr, a package from the Tidyverse used for string manipulation.

Contents

What are Regexps good for …

What are Regexps good for

Regular Expressions are very useful for retrieving patterns from a messy data set of strings. It is an alternative when manipulating strings is too much work!

All you have to do is come up with a pattern for what you want to pull out and write it using Regexps' syntax.

Using Regular Expressions with Stringr is very convenient, as many of the functions in this package use 'pattern' as an input.

What are Regexps good for

Here is an example of how Regexps are used to get the right information from a messy list of phone numbers:

```
phone_numbers <- c("(+49) 176 2692-5578",
                   "My number is 176 2678 6789",
                   "I don't have a phone",
                   "176 4567-8923")

pattern <- ".*(\\d{3}).*(\\d{4}).*(\\d{4})"

str_match(phone_numbers, pattern)
```

What are Regexps good for

Here is an example of how Regexps are used to get the right information from a messy list of phone numbers:

```
phone_numbers <- c("(+49) 176 2692-5578",
                   "My number is 176 2678 6789",
                   "I don't have a phone",
                   "176 4567-8923")

pattern <- ".*(\\d{3}).*(\\d{4}).*(\\d{4})"

str_match(phone_numbers, pattern)
```

```
##      [,1]                          [,2]  [,3]   [,4]
## [1,] "(+49) 176 2692-5578"         "176" "2692" "5578"
## [2,] "My number is 176 2678 6789" "176" "2678" "6789"
## [3,] NA                           NA    NA     NA
## [4,] "176 4567-8923"               "176" "4567" "8923"
```

Key expressions

...

Key expressions

To demonstrate the key expressions, let's use this messy list of shopping items:

```
shopping_list <- c("Apples - 5 green", "15 Oranges",
                   " green grapes - 8 package",
                   "2 packages of red grapes", "")
```

We will also use the function str_match from the package Stringr, which receives as input the string and a pattern. In our case, the pattern arguments will be the Regular Expressions.

Key expressions - Match Characters

The simplest Regular Expression you can write is the piece of string you want to match inside a quotation mark:

```
pattern <- "grapes"

str_match(shopping_list, pattern)
```

Key expressions - Match Characters

The simplest Regular Expression you can write is the piece of string you want to match inside a quotation mark:

```
pattern <- "grapes"

str_match(shopping_list, pattern)
```

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

Key expressions - Match Characters

\d matches the first digit in the string from 0 to 9. One thing to notice is that you have to add an extra \ before the expression for it to work, in order to “escape” the first \, as \ is a special character:

```
pattern <- "\\d"

str_match(shopping_list, pattern)
```

Key expressions - Match Characters

\d matches the first digit in the string from 0 to 9. One thing to notice is that you have to add an extra \ before the expression for it to work, in order to “escape” the first \, as \ is a special character:

```
pattern <- "\\d"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "5"
## [2,] "1"
## [3,] "8"
## [4,] "2"
## [5,] NA
```

Key expressions - Match Characters

\w matches the first alphanumeric element in the string (a-z, A-Z, 0–9). Here, you also have to use an extra \ to escape it:

```
pattern <- "\\w"

str_match(shopping_list, pattern)
```

Key expressions - Match Characters

\w matches the first alphanumeric element in the string (a-z, A-Z, 0–9). Here, you also have to use an extra \ to escape it:

```
pattern <- "\\w"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "A"
## [2,] "1"
## [3,] "g"
## [4,] "2"
## [5,] NA
```

Key expressions - Match Characters

\s matches the first whitespace (space, tab, newline) in the string. Again, don't forget to escape it:

```
pattern <- "\\s"

str_match(shopping_list, pattern)
```

Key expressions - Match Characters

\s matches the first whitespace (space, tab, newline) in the string. Again, don't forget to escape it:

```
pattern <- "\\s"

str_match(shopping_list, pattern)
```

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

Key expressions - Match Characters

. matches any character (wild card) that is not a void string:

```
pattern <- "."

str_match(shopping_list, pattern)
```

Key expressions - Match Characters

. matches any character (wild card) that is not a void string:

```
pattern <- "."

str_match(shopping_list, pattern)
```

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

Key expressions - Quantifiers

\+ matches 1 or more of the matching patterns in the string that were specified before:

```
pattern <- ".+"

str_match(shopping_list, pattern)
```

Key expressions - Quantifiers

\+ matches 1 or more of the matching patterns in the string that were specified before:

```
pattern <- ".+"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "Apples - 5 green"
## [2,] "15 Oranges"
## [3,] " green grapes - 8 package"
## [4,] "2 packages of red grapes"
## [5,] NA
```

Key expressions - Quantifiers

* matches 0 or more of the matching patterns in the string that were specified before. It is especially useful for finding empty strings:

```
pattern <- ".*"

str_match(shopping_list, pattern)
```

Key expressions - Quantifiers

* matches 0 or more of the matching patterns in the string that were specified before. It is especially useful for finding empty strings:

```
pattern <- ".*"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "Apples - 5 green"
## [2,] "15 Oranges"
## [3,] " green grapes - 8 package"
## [4,] "2 packages of red grapes"
## [5,] ""
```

Key expressions - Quantifiers

{n} will match exactly n of the matching patterns in the string that were specified before:

```
pattern <- ".{3}"

str_match(shopping_list, pattern)
```

Key expressions - Quantifiers

{n} will match exactly n of the matching patterns in the string that were specified before:

```
pattern <- ".{3}"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "App"
## [2,] "15 "
## [3,] " gr"
## [4,] "2 p"
## [5,] NA
```

Key expressions - Alternates

[ ] is used to match one of the characters inside of it:

```
pattern <- "[lo5]"

str_match(shopping_list, pattern)
```

Key expressions - Alternates

[ ] is used to match one of the characters inside of it:

```
pattern <- "[lo5]"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "l"
## [2,] "5"
## [3,] NA
## [4,] "o"
## [5,] NA
```

Key expressions - Alternates

| is used to match one pattern or the other:

```
pattern <- "Apple|grape"

str_match(shopping_list, pattern)
```

Key expressions - Alternates

| is used to match one pattern or the other:

```
pattern <- "Apple|grape"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "Apple"
## [2,] NA
## [3,] "grape"
## [4,] "grape"
## [5,] NA
```

Key expressions - Anchors

^ matches the beginning of a string:

```
pattern <- "^."

str_match(shopping_list, pattern)
```

Key expressions - Anchors

^ matches the beginning of a string:

```
pattern <- "^."

str_match(shopping_list, pattern)
```

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

Key expressions - Anchors

$ matches the end of a string:

```
pattern <- ".$"

str_match(shopping_list, pattern)
```

Key expressions - Anchors

$ matches the end of a string:

```
pattern <- ".$"

str_match(shopping_list, pattern)
```

```
##      [,1]
## [1,] "n"
## [2,] "s"
## [3,] "e"
## [4,] "s"
## [5,] NA
```

Key expressions - Groups

( ) is used to group the patterns inside of it:

```
pattern <- "(es).*"

str_match(shopping_list, pattern)
```

Key expressions - Groups

( ) is used to group the patterns inside of it:

```
pattern <- "(es).*"

str_match(shopping_list, pattern)
```

```
##      [,1]              [,2]
## [1,] "es - 5 green"    "es"
## [2,] "es"              "es"
## [3,] "es - 8 package"  "es"
## [4,] "es of red grapes" "es"
## [5,] NA                NA
```

How can you use Regexps to solve problems? Combinations!

...

Combinations

Consider the following list of school objects:

```
school_things <- c("notebooks 450", "sheets 1000",
                   "pencils 457", "colors 157", "nothing")
```

We will practice 5 combinations of the key expressions used before to do some interesting text manipulation. Let's start!

Combination 1

What if we wanted to get all elements from a certain kind in the school_things vector?

```
str_match(school_things, "\\d+")
```

```
##      [,1]
## [1,] "450"
## [2,] "1000"
## [3,] "457"
## [4,] "157"
## [5,] NA
```

"\\d+" reads: “R, give me one or more digits from each string”.

Combination 2

Separating numbers from words

```
new_pattern="(\\w*)\\s*(\\d*)"
str_match(school_things, new_pattern)
```

```
##      [,1]            [,2]        [,3]
## [1,] "notebooks 450" "notebooks" "450"
## [2,] "sheets 1000"   "sheets"    "1000"
## [3,] "pencils 457"   "pencils"   "457"
## [4,] "colors 157"    "colors"    "157"
## [5,] "nothing"       "nothing"   ""
```

Notice that the result is a matrix in which the columns are the vector itself, its alphanumeric values and its digits.

Combination 3

How can we find the expressions that start with not on the vector?

```
str_match(school_things, "^not.*")
```

```
##      [,1]
## [1,] "notebooks 450"
## [2,] NA
## [3,] NA
## [4,] NA
## [5,] "nothing"
```

Combination 4

Practicing conditions by finding the expressions that start with “n” or “p”

```
str_match(school_things, "[n|p].*")
```

```
##      [,1]
## [1,] "notebooks 450"
## [2,] NA
## [3,] "pencils 457"
## [4,] NA
## [5,] "nothing"
```

Combination 5

Grouping with conditions! Take the words that have two repeated vowels

```
str_match(school_things,"\\w*(a|e|i|o|u){2}.")
```

```
##      [,1]       [,2]
## [1,] "notebook" "o"
## [2,] "sheet"    "e"
## [3,] NA         NA
## [4,] NA         NA
## [5,] NA         NA
```

Where should you go to learn more & Sources

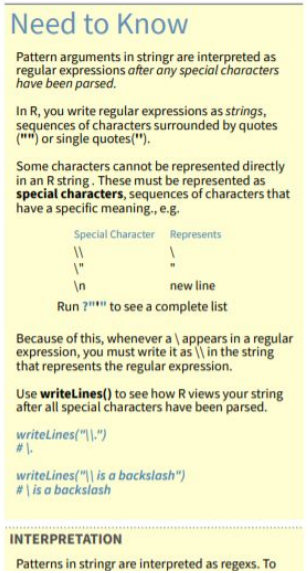
...

Where should you go to learn more

There are a bunch of cheatsheets out there. We recommend the one you can find on the Stringr website.

If you want to practice using regular expressions, you can also visit this website and use the "R Regex Texter" to double check your work!

And also, attend our tutorial right after this session :D



Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("") or single quotes(').

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run ?"'" to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

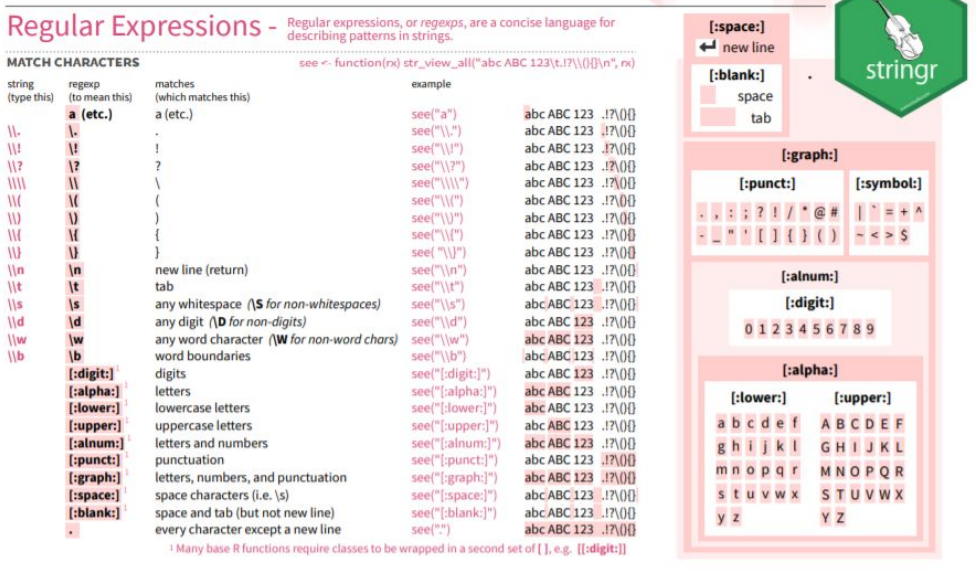
Use **writeLines()** to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.

writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs. To



Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	**\.**	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	**\!**	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	**\?**	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	****	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	**\(**	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	**\)**	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	**\{**	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	**\}**	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	**\n**	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	**\t**	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	**\s**	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	**\d**	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	**\w**	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	**\b**	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:][1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:][1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:][1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:][1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:][1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:][1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:][1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:][1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:][1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. [[:digit:]]

[:space:]
↵ new line
[:blank:]
space
tab

stringr

[:graph:]

[:punct:]
. , : ; ? ! / * @ #
- _ " ' [ ] { } ( )

[:symbol:]
| ` = + ^
~ < > $

[:alnum:]

[:digit:]
0 1 2 3 4 5 6 7 8 9

[:alpha:]

[:lower:]
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z

[:upper:]
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

Sources

1. R for Data Science
2. Automated Data Collection with R. A Practical Guide to Web Scraping and Text Mining
3. Interactive Tutorial on Regular Expressions
4. R Tutorial | Regular Expressions in R
5. Cheatsheet Stringr
6. R Regex Tester

The END...