Strings and regular expressions

Marc A.T. Teunis

2020-05-04

Packages

```r
library(tidyverse)
```

```
## -- Attaching packages -----------------------------------

## v ggplot2 3.3.0     v purrr   0.3.4
## v tibble  3.0.1     v dplyr   0.8.5
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts --------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(stringr)
```

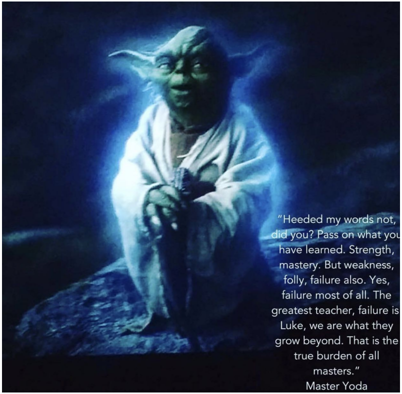
Strings and Regular Expressions

Learning aim

This lesson introduces you to character (or also called string-) manipulation in R.

- You'll learn the basics of how strings work and how to create them by hand
- The focus of this chapter will be on regular expressions, or regex for short.
- Regular expressions are useful because strings usually contain unstructured or semi-structured data
- regex are a concise language for describing patterns in strings.
- You will learn how to apply regexps to extract strings from textual or biological string data.
- In bioinformatics we often work with large strings: Think about the DNA bases in the human genome. It contains around 6.4 Billion (_6.4*10E9_) base pairs

The greatest teacher, failure is!



Yoda, The Last Jedi, 2017

Go to Rmd file –> "/demo/031_strings_and_regex.Rmd"

Strings and characters

The terminology can be confusing, therefore a short recap

```
string <- str_c("May the Force be with you!")
character_vector <- str_c("If you use a for-loop", "You are no
also_a_string <- str_c("Heeded my words not, did you? 'Pass on
empty_character <- character()

collection <- list(
  string,
  character_vector,
  also_a_string,
  empty_character
)

map(collection, str_length)
```

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

Regular $express(i)o(n)s^

When you first look at a regexp, you'll think a cat walked across your keyboard.

Complex email regex in R

```
regex_email <- str_c("[[:alnum:]\\.-_]+@[[:alnum:].-]+")
my_message <- str_c("For more information sent an email to
marc.teunis@hu.nl or
maddocent@gmail.com")
str_view_all(my_message, regex_email)
```

```
## PhantomJS not found. You can install it with webshot::insta
```

To extract the email addresses:

```r
emails <- str_match_all(my_message, regex_email)
## str_match and str_match_all, return a matrix
as_tibble(unlist(emails))
```

```
## # A tibble: 2 x 1
##   value
##   <chr>
## 1 marc.teunis@hu.nl
## 2 maddocent@gmail.com
```

Prerequisites

This lesson will focus on the {stringr} package for string manipulation. stringr is not part of the core tidyverse because you don't always have textual data, so we need to load it explicitly.

```r
library(stringr)
```

String basics

- Create strings with either single quotes or double quotes.
- Unlike other languages, there is no difference in behaviour.
- I recommend always using ", unless you want to create a string that contains multiple ".

```
string1 <- "This is a DNA string: AAAGGCGCGAAGG"
string2 <-
"If I want to include a quote: 'AAAGG' inside a string, I use
string3 <- 'But you can also create a string and quote "AAGG"
```

Every "quote" needs a friend

If you forget to close a quote, you'll see +, the continuation character:

```
> "This is a string with an opening quote that does not have a
+
+
+ HELP I'M STUCK
```

If this happen to you, press Escape and try again!

Escape with "\"

To include a literal single or double quote in a string you can use \ to "escape" it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

That means if you want to include a literal backslash, you'll need to double it up: `"\\"`. To see the raw contents of the string, use `writeLines()`:

```
x <- c("\"", "\\")
x
```

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

```
## To see what the escaped string translates to
writeLines(x)
```

```
## "
```

Special characters

- There are a handful of other special characters. The most common are `"\n"`, newline, and `"\t"`, tab, but you can see the complete list by requesting help on ": `?'"'`, or `?"'"`.
- Strings like `"\u00b5"`, are a way of writing non-English characters that work on all OS platforms:

```
micro <- "\u00b5"
micro
```

```
## [1] "µ"
```

```
writeLines(micro)
```

```
## µ
```

String length

- Base R contains many functions to work with strings but we'll avoid them because they can be inconsistent
- We'll use functions from {stringr}.
- These have more intuitive names, and all start with `str_`.
- For example, `str_length()` tells you the number of characters (of each element) in a string (or character vector):

```
str_length(c("Autobots!", "transform", "and roll out!"))
```

```
## [1]  9  9 13
```

Combining strings

To combine two or more strings, use `str_c()`:

```
str_c("My favorite autobot is", "Perceptor")
```

```
## [1] "My favorite autobot isPerceptor"
```

Use the `sep` argument to control how they're separated:

```
str_c("Autobot", "Perceptor", sep = "[::..::]")
```

```
## [1] "Autobot[::..::]Perceptor"
```

Vectorization

As shown above, `str_c()` is vectorised, and it automatically recycles shorter vectors to the same length as the longest:

```r
str_c("prefix-", c("a", "b", "c"), "-suffix")
```

```
## [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Collapse a vector of multiple strings

Use `collapse`:

```
str_c(c("x", "y", "z"), collapse = "/")
```

```
## [1] "x/y/z"
```

Subsetting strings

You can extract parts of a string using `str_sub()`. As well as the string, `str_sub()` takes `start` and `end` arguments which give the (inclusive) position of the substring:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

```
# negative numbers count backwards from the end of a string
str_sub(x, -3, -1)
```

```
## [1] "ple" "ana" "ear"
```

Note that `str_sub()` won't fail if the string is too short: it will just return as much as possible:

```
str_sub("a", 1, 5)
```

Type case

- To change the type face from capitals to lower case us `str_to_lower()`
- You can also use `str_to_upper()` or `str_to_title()`.

Section 1

Regular Expressions

Matching patterns with regular expressions

- Regexps are a very terse language that allow you to describe patterns in strings.
- Once you understand them, you'll find them extremely useful.
- Remember Yoda: **"The greatest teacher, failure is!"**

"str_view" and "str_view_all"

To learn regular expressions, we'll use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression, and show you how they match.

We'll limit ourselves to simple regular expressions and pattern matching in this demo

Basic matches

The simplest patterns match exact strings: `str_view()` shows the **first** match

```
x <- c("AAAAGGCGC", "CCCGCGAATTT", "TTTCGCGCGCG")
str_view(x, "GC")
```

What do you notice on the match in the last string?

Match all

```r
x <- c("AAAAGGCGC", "CCCGCGAATTT", "TTTCGCGCGCG")
str_view_all(x, "GC")
```

The next step up in complexity is "."

This matches any character (except a newline):

```
str_view_all(x, ".A.")
```

```
str_view_all(x, "(.A|C.)") ## pattern ending with A or startin
```

Using (escapes) in pattern matching regex

If "." matches any character, how do you match the character "."?

- You need to use an "escape" to tell the regular expression you want to match it exactly, not use its special behaviour.
- To match an `.`, you need the regexp `\.`.
- We use strings to represent regular expressions, and \ is also used as an escape symbol in strings. *To create the regular expression `\.` we need the string `"\\."`.

```
# To create the regular expression, we need \\
dot <- "\\."
# But the expression itself only contains one dot:
writeLines(dot)
```

```
## \.
```

And this tells R to look for an explicit dot

```r
str_view(c("abc", "a.c", "feg"), "\\.c")
```

```r
writeLines("\\.c")
```

```
## \.c
```

Anchors

By default, regular expressions will match any part of a string. It's often useful to *anchor* the regular expression so that it matches from the start or end of the string. You can use:

- ^ to match the start of the string (head)
- $ to match the end of the string (tail)

"If you start with power, you will end up with money" - Jenny Brian, 2015, CSAMA Course, Italian Alpes

If you start with power (^), you end up with money ($)

```
x <- c("AAATTCCC", "AAATATT", "AACGGCGCA")
str_view(x, "^A")
```

```
str_view(x, "A$")
```

To force a regular expression to only match a complete string, anchor it with both ^ and $:

```
x <- c("AAATATTCCC", "AAATATT", "AACGGCGCA")
str_view(x, "AAATATT")
```

```
str_view(x, "^AAATATT$")
```

Character classes and alternatives

There are a number of special characters to use in matching:

- `\d`: matches any digit.
- `\s`: matches any whitespace (e.g. space, tab, newline).
- `[abc]`: matches a, b, or c.
- `[^abc]`: matches anything except a, b, or c.
- `[:alnum:]`: matches all numbers AND digits (alpha-numericals)

Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the \ for the string, so you'll type `"\\d"` or `"\\s"`.

Alternation to choose between patterns

- Use *alternation* to pick between one or more alternative patterns.
- `abc|d..f` will match either ‘“abc”’, or `"deaf"`, `doof`, `deef`,`daaf`, etc.
- The precedence for `|` is low, so that `abc|xyz` matches `abc` or `xyz` not `abcyz` or `abxyz`.

If precedence (which comes first?) ever gets confusing, use parentheses to make it clear what you want:

```
str_view(c("grey", "gray", "greay"), "gr(e|a)y")
```

Repetition

The next step up in power involves controlling how many times a pattern matches:

- ?: 0 or 1
- +: 1 or more
- *: 0 or more

```
x <- c("AAATATACCAA", "CCCGCGCGC", "AAATCCCCGGCGCCC")
str_view_all(x, "CCC?") ## ? (question mark) : 0 or 1 occurren
```

```
str_view_all(x, "CC+") ## :  1 or more occurrences
```

```
str_view_all(x, "C{3}") ##  : exactly {n} times (can be any nu
```

Combine things

```r
str_view(x, "[C].+A*")
```

Precendence of repetition

Note that the precedence of these operators is high, so you can write: `colou?r` to match either American or British spellings. That means most uses will need parentheses, like `bana(na)+`.

The previous "banana" example repeated

```
fruits <- c("apple", "banana", "pear")
str_view_all(fruits, ".a.+")
```

```
str_view_all(fruits, ".(a.)+")
```

```
str_view_all(fruits, "(.a).+")
```

Specify the number of matches

- `{n}`: exactly n
- `{n,}`: n or more
- `{,m}`: at most m
- `{n,m}`: between n and m

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVII
str_view(x, "C{2}")
```

```
str_view(x, "C{2,}")
```

```
str_view(x, "C{2,3}")
```

Grouping and backreferences

Earlier, you learned about parentheses as a way to disambiguate complex expressions. They also define "groups" that you can refer to with *backreferences*, like \1, \2 etc. For example, the following regular expression finds all fruits that have a repeated pair of letters.

```
str_view(fruit, "(..)\\1", match = TRUE)
```

Section 2

Regex tools

What can you do with regex tools?

With the {stringr} package you can:

- Determine which strings match a pattern.
- Find the positions of matches.
- Extract the content of matches.
- Replace matches with new values.
- Split a string based on a match.

Be aware that regular expressions can become quite complex, really fast. A word of caution before we continue: because regular expressions are so powerful, it's easy to try and solve every problem with a single regular expression. In the words of Jamie Zawinski:

"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."

As a cautionary tale, check out this regular expression that checks if a email address is valid:

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

Detect matches

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input:

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
```

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

- When you use a logical vector in a numeric context, `FALSE` becomes 0 and `TRUE` becomes 1.
- That makes `sum()` and `mean()` useful if you want to answer questions about number of matches across a larger vector:

"str_detect()" examples

```r
# How many common words start with t?
sum(str_detect(words, "^t"))
```

```
## [1] 65
```

```r
# What proportion of common words end with a vowel?
mean(str_detect(words, "[aeiou]$"))
```

```
## [1] 0.2765306
```

A common use of `str_detect()` is to select the elements that match a pattern. You can do this with logical subsetting, or the convenient `str_subset()` wrapper:

str_subset()

```r
words[str_detect(words, "x$")]
```

```
## [1] "box" "sex" "six" "tax"
```

```r
str_subset(words, "x$")
```

```
## [1] "box" "sex" "six" "tax"
```

Match in a dataframe

Typically, however, your strings will be one column of a data frame, and you'll want to use filter instead:

```
df <- tibble(
  word = words,
  i = seq_along(word) ## i is an index indicating the row pos
)
df %>%
  filter(str_detect(words, "x$"))
```

```
## # A tibble: 4 x 2
##   word      i
##   <chr> <int>
## 1 box     108
## 2 sex     747
## 3 six     772
## 4 tax     841
```

Counting matches in a string

```
x <- c("CCCGGTTTAAAATGTAACCGCGTAG", "AAAATATGGGTGTGATGATGTAGTA
str_view_all(x, "ATG")
```

```
str_count (x, "ATG")
```

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

```
str_view_all(x, "TAG")
```

```
str_count(x, "TAG")
```

```
## [1] 1 2
```

Extract matches

- To extract the actual text of a match, use `str_extract()`.
- Use the Harvard sentences, which were designed to test VOIP systems, but are also useful for practicing regexps.
- These are provided in `stringr::sentences`:

```
length(sentences)
```

```
## [1] 720
```

```
head(sentences)
```

```
## [1] "The birch canoe slid on the smooth planks."
## [2] "Glue the sheet to the dark blue background."
## [3] "It's easy to tell the depth of a well."
## [4] "These days a chicken leg is a rare dish."
## [5] "Rice is often served in round bowls."
## [6] "The juice of lemons makes fine punch."
```

Find all sentences that contain a certain colour

Imagine we want to find all sentences that contain a colour. We first create a vector of colour names, and then turn it into a single regular expression:

```
colours <- c("red", "orange", "yellow", "green", "blue", "purp
colour_match <- str_c(colours, collapse = "|")
colour_match
```

```
## [1] "red|orange|yellow|green|blue|purple"
```

Extract the colours

Now we can select the sentences that contain a colour, and then extract the colour to figure out which one it is:

```
has_colour <- str_subset(sentences, colour_match)
head(has_colour)
```

```
## [1] "Glue the sheet to the dark blue background."
## [2] "Two blue fish swam in the tank."
## [3] "The colt reared and threw the tall rider."
## [4] "The wide road shimmered in the hot sun."
## [5] "See the cat glaring at the scared mouse."
## [6] "A wisp of cloud hung in the blue air."
```

To get all matches, use `str_extract_all()`. It returns a list:

```
head(str_extract_all(sentences, colour_match))
```

Replacing matches

`str_replace()` and `str_replace_all()` allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

Multiple replacements

With `str_replace_all()` you can perform multiple replacements by supplying a named vector:

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
```

```
## [1] "one house"    "two cars"     "three people"
```

Splitting

Use `str_split()` to split a string up into pieces. For example, we could split sentences into words:

```
sentences %>%
  head(5) %>%
  str_split(" ")
```

```
## [[1]]
## [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"
## [8] "planks."
##
## [[2]]
## [1] "Glue"        "the"         "sheet"       "to"
## [6] "dark"        "blue"        "background."
##
## [[3]]
## [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"
```

Splitting up to pieces

You can also request a maximum number of pieces:

```
fields <- c("Name: Marc", "Country: NL", "Age: 46")
fields %>% str_split(": ", n = 2, simplify = TRUE)
```

```
##      [,1]      [,2]
## [1,] "Name"    "Marc"
## [2,] "Country" "NL"
## [3,] "Age"     "46"
```

Find matches

str_locate() and str_locate_all() give you the starting and ending positions of each match. These are particularly useful when none of the other functions does exactly what you want. You can use str_locate() to find the matching pattern, str_sub() to extract and/or modify them.

```
dna <- str_c("AAAAAGCGGGCGC")
ind <- str_locate_all(dna, "AA")
str_sub(dna, start = ind[[1]][1,1], end = ind[[1]][1,2])
```

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